문제해결을 위한

창의적 알고리즘(고급)

■ 집필진

안성진(성균관대학교 교수)
송태옥(가톨릭관동대학교 교수)
장승연(성균관대학교 연구원)
정종광(경기과학고등학교 교사)
배준호(경남정보고등학교 교사)
김봉석(경남과학고등학교 교사)
오은희(창원과학고등학교 교사)
정혜진(경기과학고등학교 교사)
전현석(경기과학고등학교 교사)
문광식(세종특별자치교육청 교사)
장원영(충북교육정보원 교사)
최웅선(수원정보과학고등학교 교사)
이 진(인천과학고등학교 교사)
박병기(서울과학고등학교 교사)
김종혜(경기과학고등학교 교사)
한건우(경기모바일과학고등학교 교사)
정웅열(경기북과학고등학교 교사)
배성환(광주과학고등학교 교사)
박소영(부산일과학고등학교 교사)

▌목 차▌

Ⅰ. 관계기반 알고리즘의 설계

1. 수학적 귀납법과 점화식 7
   가. 수학적 귀납법 7
   나. 귀납법을 이용한 재귀함수의 설계 8
2. 동적표를 이용한 알고리즘 설계 71
   가. 하향식 설계 76
   나. 상향식 설계 79
3. 동적표를 이용한 중급 기법 199

Ⅱ. 알고리즘 설계기법의 응용

4. 이분탐색을 활용한 설계기법 217
5. 자료구조를 활용한 알고리즘의 고속화 238

Ⅲ. 전 세계적 온라인 대회 참가하기

6. USACO Online Competition 271
   가. 가입 방법 272
   나. 비밀번호 재설정하기 273
   다. USACO Competition 274
   라. USACO Competition 참가하기 275
   마. USACO Competition 기출문제 풀어보기 278
7. Codeforces 282
   가. Codeforces에 가입하기 282
   나. 대회에 참가하기 284
   다. 대회에서 점수 산정 방법 288

Part

I

관계기반 알고리즘의 설계

1. 수학적 귀납법과 점화식
2. 동적표를 이용한 알고리즘 설계
3. 동적표를 이용한 중급 기법

Part I

관계기반 알고리즘의 설계

이번 단원에서는 관계기반 설계방법에 대해서 알아본다.

중급 편에서는 탐색기반의 설계방법에 대해서 알아보았다. 탐색기반의 설계방법은 컴퓨터의 빠른 연산속도를 이용하여 짧은 시간에 가능한 해의 집합을 탐색하면서 최적해를 구하는 기술적인 방법이라면, 관계기반 설계방법은 해를 구하는 행위를 하나의 함수로 표현하고 이 함수들의 관계를 이용하여 해를 구하는 아주 효율적인 방법이다.

관계기반 설계를 적용하기 위해서는 문제의 정의 및 상태를 함수로 정의하고 이 함수들 간의 관계를 점화식 혹은 이와 유사한 형태로 표현할 수 있어야 한다. 관계기반 설계에서는 수학적 귀납법과 점화식 등의 표현이 기반이 되므로 이번 단원에서는 수학적 귀납법에 대해서 간단히 살펴보고 관계기반으로 알고리즘을 설계하는 방법에 대해서 다룬다.

매우 복잡하고 어려워 보이는 문제도 관계기반 설계로 간단히 해결되는 경우가 있다. 이러한 해법들은 보통 아주 간결한 형태로 표현되지만 처음부터 어려운 문제들을 관계기반으로 설계하기란 쉽지가 않다.

이번 단원에서는 아주 쉬운 문제부터 단계별로 관계기반 설계를 적용하는 방법에 대해서 연습한다.

1 수학적 귀납법과 점화식

가. 수학적 귀납법

수학적 귀납법은 자연수 n에 관한 명제 $P(n)$이 모든 자연수에 대해서 성립함을 증명하기 위한 수학의 증명법 중 한 방법이다.

자연수 n에 관한 명제 $P(n)$이 모든 자연수 n에 대해 성립함을 다음과 같은 2가지 단계로 증명한다.

① $P(1)$이 성립함을 보인다 - Basis
② $P(k)$가 성립한다고 가정하고 $P(k+1)$이 성립함을 보인다 - induction

①, ②를 통해서 모든 자연수에 대해서 $P(n)$이 성립함을 보이는 방법이다.

단계별로 적용하면 왜 모든 자연수에 대해서 명제 $P(n)$이 성립하는지 쉽게 알 수 있다. 다음 단계를 밟아가며 성립함을 알아보자.

$P(1)$은 ①에서 성립함을 보였으므로, $P(2)$가 성립함을 알아보자. ②에서 $P(1)$이 성립한다면 $P(2)$가 성립함을 보였다. 따라서 $P(1)$이 성립함은 이미 증명되었으므로 $P(2)$도 증명된다. 이와 같은 식으로 $P(k)$에 대해서도 모든 경우 성립함을 보일 수 있다.

수학적 귀납법은 정보과학에서 문제를 해결하는 데 있어서 매우 중요한 요소이다. 정보과학에서는 수학적 귀납법으로 명제를 증명할 때, 모든 자연수에 대해서 증명하는 것이 아니라, 문제의 입력이 n이고 문제에서 구하는 해가 $f(n)$일 때, 귀납적 관계를 이용하여 $f(n)$을 구할 때 사용한다.

수학적 귀납법을 이용한 문제해결의 절차는 다음과 같다.

① 입력값이 n인 문제의 해를 $f(n)$으로 정의한다.
② $f(1)$을 직접 구하여 출력한다.
③ $f(k)$를 이미 구해두었다고 가정하고 $f(k)$를 통하여 $f(n)$을 구하고 출력한다.

여기서 k는 $n-1$이 될 수도 있고, $\frac{n}{2}$이 될 수도 있다. 여기서 중요한 것은 k를 문제에 따라 적절한 형태로 설계해야 한다는 점이다.

나. 귀납법을 이용한 재귀함수의 설계

문제를 귀납적으로 해결할 때는 일반적으로 재귀함수를 이용한다. 다음 예제를 통하여 귀납적 설계방법을 익히자.

예

임의의 정수 n을 입력받아 1부터 n까지 합을 구하는 프로그램을 작성하시오.
(단, $n \le 100$)

풀이 1

```
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

일단 먼저 함수 $f(n)$을 다음과 같이 정의하자.

$$f(n) = \text{'1부터 } n\text{까지의 합'}$$

다음으로 $f(1)$을 정의에 의하여 직접 구하면

$$f(1) = 1$$

$f(n-1)$을 이미 구해두었다고 가정하면, 다음과 같이 $f(n)$을 구할 수 있다. $f(10)$을 $f(9)$를 이용하여 나타내보자.

$$f(10) = 1+2+3+4+5+6+7+8+9+10$$
$$f(9) = 1+2+3+4+5+6+7+8+9$$

따라서 $f(10) = f(9) + 10$ 과 같다. 이를 일반화하면,

$$f(n) = f(n-1) + n$$

이 관계를 이용하여 재귀함수를 작성하면 다음과 같다.

줄 | 코드 | 참고

```c
#include <stdio.h>

int f(int n)
{
  if(n==1) return 1;
  return f(n-1)+n;
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

풀이 2

$f(n)$과 $f(k)$의 관계를 다르게 정의해보자. 이번에는 $k=\frac{n}{2}$으로 정의해본다.

주어진 입력값이 10이라고 생각하면, $f(10)$을 구하는 문제이다.
$f(10)$과 $f(5)$를 각각 나타내면 다음과 같다.

$$f(10) = 1+2+3+4+5+6+7+8+9+10$$
$$f(5) = 1+2+3+4+5$$

다시 $f(10)$을 $f(5)$를 이용하여 표현하면,

$$\begin{aligned} f(10) &= f(5) + 6+7+8+9+10 \\ &= f(5) + (5+1)+(5+2)+(5+3)+(5+4)+(5+5) \\ &= f(5) + f(5) + (5\times5) \\ &= 2f(5)+5^2 \end{aligned}$$

즉 이를 일반화하면,

$$f(n) = 2f(\frac{n}{2}) + (\frac{n}{2})^2$$

이 된다. 단, 위 관계식은 n이 짝수일 때만 성립된다. 만약 n이 홀수라면 다음 식을 얻을 수 있다.

$$f(n) = 2f(\left[\frac{n}{2}\right]) + (\frac{n+1}{2})^2$$

이 된다. 앞 두 식을 홀, 짝에 관계없이 쓸 수 있도록 일반화하면,

$$f(n) = 2f(\left[\frac{n}{2}\right]) + (\left[\frac{n+1}{2}\right])^2$$

이 된다. 프로그래밍 언어에서는 정수를 나누면 소수점 이하를 버리므로 다음과 같은 알고리즘을 작성할 수 있다.

코드	참고

```c
#include <stdio.h>

int f(int n)
{
  if(n==1) return 1;
  return 2*f(n/2)+((n+1)/2)*((n+1)/2);
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

앞 두 예와 같이 관계식을 설계하는 방법에 따라 다양한 알고리즘을 설계할 수 있는 것

이 특징이며, 설계방법에 따라 계산량이 달라진다. 참고로 [풀이 1]의 계산량이 $O(n)$ 이고, [풀이 2]의 계산량은 $O(\lg\ n)$[1]이다. 이와 같이 최대한 효율적인 설계를 할 수 있도록 연습하는 것이 중요하다.

이러한 문제해결 방법을 분할정복(divide and conquer)기법이라고 한다. 일반적으로 분할정복 기법이라고 함은 입력크기 n을 동일한 크기의 k개의 함수로 분할하여 이 분할의 결과를 이용하여 전체 문제를 해결하는 방법을 말한다. [풀이 1]과 같은 방법은 $n-1$과 1로 나누었기 때문에 효율은 떨어질 수 있지만 넓은 의미로 분할정복이라고 할 수 있다.

이번에는 수학 시간에 흔히 다루던 진법변환에 대해서 알아보자. 다음 문제를 통해 재귀함수의 강력함을 익힐 수 있도록 하자.

예

임의의 10진법 정수 n을 입력받아 이를 k진법으로 출력하는 프로그램을 작성하시오.
(단, $n \le 100{,}000$; $k \le 20$)

풀이

진법 변환에 대해서 알아보자. 만약 십진법의 수 12를 이진법으로 고치려면 다음과 같은 단계를 거친다.

```
1 2
  2
----
  6 …    0
```

12를 2로 나누고 그 몫은 6이고 나머지는 0이다.
여기서 나머지인 0은 오른쪽에 따로 써 둔다.

1) $\lg n = \log_2 n$

6				위 연산의 몫인 6을 이용하여 다시 2로 나눈다. 이 때 나머지는 0이고 몫은 3을 각각 같은 방법으로 쓴다.
2				
3	…	0		

3				마찬가지로 3을 2로 나눈 몫과 나머지를 기록한다.
2				
1	…	1		

1	2				마지막의 몫이 2보다 작으므로 더 이상할 필요가 없다.
	2				
	6	…	0	④	이 때 맨 아래 2보다 작은 1인 몫과 나머지를 아래에서부터 위로 하나씩 ①로부터 ④까지 연결한
	2				
	3	…	0	③	1100
	2				
	1	…	1	②	이라는 값이 12를 이진법으로 바꾼 값이 된다.
	①				

이와 같은 방법은 반복문으로 쉽게 구현이 가능하다. 하지만 관계식으로 보다 명확하게 표현할 수 있다.

십진법의 수 n을 k진법으로 나타내는 함수를 다음과 같이 정의해보자.

$f(n,\ k)$ = 10진법의 수 n을 k진법으로 출력함

앞과 같이 표현하고, 앞 표현을 귀납적으로 완성하기 위하여 재귀호출 없이 직접 구현하는 부분이 있어야 하므로, 다음과 같이 정의한다.

$$f(n,\ k)\ =\ n\ (\text{단},\ n<k)$$

혹은 다음과 같이 나타낼 수도 있다.

$$f(n,\ k)\ =\ 0\ \text{혹은 아무것도 출력하지 않음.}\ (\text{단},\ n=0)$$

마지막으로 관계를 정리해보자. $f(n,\ k)$에서 n을 k로 나눈 몫과 나머지가 필요하며, 몫은 다음 연산의 n값으로 사용되며, 나머지는 출력한다.

여기서 중요한 점은 나머지가 거꾸로 출력되어야 하므로 먼저 $f(n/k,\ k)$의 결과가 다 출력되었다고 가정한 상태에서 다음에 $n\%k$를 출력하면 그 결과는 높은 자리의 수부터 정확하게 출력된다.

생각해보면 당연한 결과이다. n을 k로 나눈 몫을 k진법으로 이미 바꾸어 놓았다고 가정하자. 예를 들어 12를 이진법으로 고칠 때, 6을 이미 이진법으로 고쳐서 110을 출력해 두었다고 가정하자.

이제 이 결과 뒤에 12를 2로 나눈 나머지인 0을 출력하면 110에 0을 붙여 1100이 된다. 따라서 정확하게 처리됨을 알 수 있다. 이 때 6이 어떻게 110으로 출력되는 지에는 관심을 가질 필요가 없다. 왜냐하면 이미 6이 이진법으로 출력되어 있음은 귀납적으로 가정되었기 때문이다.

따라서 관계를 정리하면 다음과 같은 식을 유도할 수 있다.

$$f(n,\ k)\ =\ f(n/k,\ k)\ \text{를 출력한 후},\ print(n\%k)$$

이 관계를 이용하여 진법변환을 쉽게 표현할 수 있다. 앞으로 "~를 한 후"의 표현을 그냥 ","로 나타내고자 한다. 위 식들을 모두 정리하면 다음과 같은 관계식을 만들 수 있다.

$$f(n,\ k)\ =\ \begin{cases} n & (n<k) \\ f(n/k,\ k),\ print(n\%k) & (n\ge k) \end{cases}$$

마지막으로 k 진법으로 출력하고자 할 때, k 값이 10을 초과하면 출력할 문자가 알파벳으로 바뀌어야 한다. 이를 처리하는 방법은 문자열을 이용할 것이므로 참고하기 바란다. 이 방법은 앞으로 많은 문제에 응용되므로 꼭 익혀두기 바란다.

위 관계식을 구현한 소스코드는 다음과 같다.

```
#include <stdio.h>

void f(int n, int k)
{
  if(n<k) printf("%d", n);
  f(n/k, k), printf("%d", n%k);
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  f(n, k);
  return 0;
}
```

위 방법으로 구현하면 10진법까지는 모두 바꿀 수 있다. 다만 알파벳을 출력하는 부분이 없으므로 11진법 이상은 잘못된 값이 출력된다.

“%x”라는 형식지정자를 이용하여 다음과 같이 바꿀 수 있다.

```
#include <cstdio>

void f(int n, int k)
{
  if(n<k) printf("%x", n);
  f(n/k, k), printf("%x", n%k);
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  f(n, k);
  return 0;
}
```

이와 같이 바꾸면 16진법까지는 처리가능하다. 하지만 이 문제에서는 20진법까지 바꿀 수 있어야하므로, 다음과 같이 문자열을 이용해보자.

```
#include <cstdio>

char d[21] = "0123456789ABCDEFGHIJ";

void f(int n, int k)
{
  if(n<k) printf("%c", d[n]);
  else f(n/k, k), printf("%c", d[n%k]);
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  f(n, k);
  return 0;
}
```

이와 같이 변경하면 매우 효율적으로 20진법까지 표현 가능하다. 만약 이를 if문이나 switch~case문을 이용한다면 코드가 훨씬 길어지고 복잡해질 것이다.

마지막으로 관계설정 없이 직접 구현하는 코드를 다음으로 변경하면 코드의 길이를 최소화할 수 있다.

$f(n,\ k)\ =\ 0$ 혹은 아무것도 출력하지 않음. (단, $n=0$)

위 코드를 이용하여 다음 코드와 같이 진법변환을 최소화하여 표현할 수 있다.

코드	참고

```
#include <cstdio>

char d[21] = "0123456789ABCDEFGHIJ";

void f(int n, int k)
{
    if(n>0) f(n/k, k), printf("%c", d[n%k]);
}

int main()
{
    int n, k;
    scanf("%d %d", &n, &k);
    f(n, k);
    return 0;
}
```

함수 f의 핵심을 한 줄로 표현하고 있다. 이와 같이 재귀함수와 관계기반 설계를 적절히 적용하면 복잡해 보이는 식을 매우 간략하게 표현할 수 있으므로, 작성하기 쉬운 알고리즘 설계법이다.

다음 주어진 문제들을 통하여 관계기반 설계의 기본을 익혀보자.

문제 1

숫자 뒤집기(S)

하나의 정수가 입력된다.

이 정수를 거꾸로 출력하는 프로그램을 작성하시오.

예를 들어 입력되는 정수가 123이라면 321을 출력하면 된다.

단, 12300이 입력될 경우 00321을 출력하는 것이 아니라 321을 출력해야 함에 주의해야 한다.

입력

첫 줄에 하나의 정수가 입력된다.
(1 <= n <= 50,000)

출력

입력된 정수를 거꾸로 출력한다.

입력 예	출력 예
123	321
12300	321

풀이

이 문제는 초급에서 자주 다루는 간단한 문제이다. 이 문제를 해결하는 다양한 방법이 있지만 이 단원에서는 관계기반으로 설계해보자.

먼저 $f(n)$을 직접 값으로 정의하지 않고 $solve(n)$을 이용한 재귀호출을 이용하여 단순히 해결하는 방법을 알아보자.

① n을 입력받는다.
② $n\%10$을 출력한 후 $solve\left(\left[\frac{n}{10}\right]\right)$을 호출한다.
③ n이 0이 아니면 ②의 과정으로 이동한다.

$solve(123)$의 과정을 보면 다음과 같이 동작한다.

3을 출력한 후 $solve(12)$
2를 출력한 후 $solve(1)$
1을 출력한 후 $solve(0)$

따라서 위와 같은 단계를 거치면서 화면에는 321이 출력될 것이다.

이 방법을 소스코드로 작성한 예는 다음과 같다.

```
#include <cstdio>

void solve(int n)
{
  if(n==0) return;
  printf("%d", n%10);
  solve(n/10);
}

int main()
{
  int n;
  scanf("%d", &n);
  solve(n);
  return 0;
}
```

이와 같이 작성하면 정확하게 거꾸로 출력된 수를 볼 수 있을 것이다. 여기서 주의할 점은 12300을 00321으로 출력한다는 점이다.

즉, 거꾸로 할 때 맨 앞에 0은 생략해야 하므로 코드를 다음과 같이 수정해야 한다.

코드	참고

```
#include <cstdio>

bool first=1;

void solve(int n)
{
  if(n==0) return;
  if(first && (n%10)==0)
    solve(n/10);
  else
  {
    printf("%d", n%10), first=0;
    solve(n/10);
  }
}

int main()
{
  int n;
  scanf("%d", &n);
  solve(n);
  return 0;
}
```

여기서 first는 첫 번째 숫자를 출력할지의 여부를 결정하는 불변수이다. 첫 번째 변수가 0이라면 출력하지 않고 무시하는 부분을 포함하고 있다.

이번에는 새로운 방법으로 관계를 만들어 보자. 이번 방법은 다소 수학적인 아이디어가 필요한 방법이다.

먼저 $f(n)$을 다음과 같이 정의하자.

$f(n)$ = "n을 거꾸로 출력한 수"

이번에는 $f(n)$ 이 함수이므로 관계식을 만들어야 한다. 먼저 n 이 10 미만인 경우에는 거꾸로 써도 같은 값이므로 이 값들은 관계식 없이 직접 구할 수 있다.

$$f(n) = n \quad (n < 10)$$

다음으로 n 이 10이상일 경우에 대해서 살펴보자. 실제 예로 $n = 123$ 이라면 $f(n) = 321$ 이어야 한다. 이 과정을 자세히 분석하면 다음과 같다.

① n 의 1의 자릿수와 나머지를 분리 한다. ($\frac{n}{10} = 12$, $n \% 10 = 3$)
② $\frac{n}{10}$ 을 거꾸로 만든다. 즉 $f(\frac{n}{10})$ 을 취한다. (21)
③ 이 값에 $n \% 10$ 의 값에 $10^{(123\text{의 자릿수} - 1)}$ 을 곱한다. (300)
④ ②, ③에서 구한 값을 더한다. (321)

이와 같은 과정으로 123을 321 로 바꿀 수 있다. 이 경우에는 산술연산이 적용되므로 맨 앞자리 0 은 자동적으로 없어진다. 여기서 중요한 점은 n 의 자릿수를 나타내는 값을 구하는 방법이다.

이 값은 log 함수를 이용하면 쉽게 구할 수 있다. math.h에 상용로그($\log_{10}()$)와 자연로그($\log()$)의 함수를 제공하고 있으므로 상용로그를 이용하자. 위 결과를 관계식으로 일반화 하여 표현하면 다음과 같다.

$$f(n) = \begin{cases} n & (n < 10) \\ f(\frac{n}{10}) + (n\%10) \times 10^{[\log n]} & (n \geq 10) \end{cases}$$

이 관계식을 이용하여 소스코드로 구현할 수 있다.

코드	참고

```
#include <cstdio>
#include <cmath>

int f(int n)
{
  if(n<10) return n;
  return f(n/10)+(n%10)*powf(10.0,(int)log10((double)n));
```

줄	코드	참고
8	`}`	
9		
10	`int main()`	
11	`{`	
12	`  int n;`	
13	`  scanf("%d", &n);`	
14	`  printf("%d\n", f(n));`	
15	`  return 0;`	
16	`}`	

이 구현에서 7행의 각종 수학 관련 함수들의 활용법을 잘 익힐 수 있도록 한다. 여기서 사용한 함수들의 의미는 다음과 같다.

> $powf(x,y)$: float x, y에 대해서 x^y를 구한다.
> $\log_{10}(x)$: double x에 대해서 $\log x$를 구한다.

마지막으로 관계식을 조금 더 개량해 보자.

이번에는 다음과 같은 아이디어로 관계식을 유도하자.

> ① n의 1의 자릿수를 분리한다.
> ② 가장 큰 자릿수를 분리한다.
> ③ 1의 자리와 가장 큰 자리를 뺀 m에 대해 $f(m)$ 을 구한다.
> ④ ①$\times 10^{[\log n]}$ + $f(m)$ + ②를 구하면 된다.

위의 관계 설계 아이디어를 식으로 나타내면 다음과 같다. 단, $T = 10^{[\log n]}$이다.

$$f(n) = \begin{cases} n & (n < 10) \\ T(n\%10) + 10f(\frac{n\%T}{10}) + [\frac{n}{T}] & (n \geq 10) \end{cases}$$

좀 복잡해 보이긴 하지만 원리를 생각해보면 간단하다. 이를 소스코드로 구현한 결과는 다음과 같다.

줄	코드	참고

```
#include <cstdio>
#include <cmath>

int f(int n)
{
  if(n<10) return n;
  int T=(int)powf(10.0,((int)log10(n)));
  return (n%10)*T+f((n%T)/10)*10+n/T;
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

문제 2

별 그리기

한 정수 n을 받아서 n층의 삼각형 모양의 별을 출력하시오.
(단 1 <= n < 10,000)

아래 그림은 n이 3인 경우이다.

```
*
**
***
```

입력

첫 번째 줄에 정수 n이 입력된다.
(단, 1 <= n < 10,000)

출력

n줄에 걸쳐서 i번째 줄에 i개의 "*"를 출력한다.

입력 예	출력 예
3	* ** ***
5	* ** *** **** *****

풀이

이 문제는 반복문 활용 예제로 자주 다루던 문제이다. 이 문제를 관계기반 설계법으로 해결해보자.

먼저 문제를 정의해야 한다. 이 문제는 간단하게 정의할 수 있다.

$f(n)$ = "i 행에 i 개의 '*'이 n 행에 걸쳐서 그려진 패턴" (단, $1 \le i \le n$)

이제 각 항목들 간의 관계를 정의하기 위해서 각 상태들을 관찰하자.

정의에 의하여 $f(1)$ 부터 $f(4)$ 까지의 패턴은 다음과 같다.

```
f(1) =   *     f(2) =  *     f(3) =  *       f(4) =  *
                       **            **              **
                                     ***             ***
                                                     ****
```

이 패턴들을 통하여 규칙을 찾아보자. 다음으로 $f(5)$ 를 그리고자 할 때, 만약 $f(4)$ 가 이미 그려져 있다고 가정하고, 이로부터 $f(5)$ 를 그리면 다음과 같다.

```
f(4) =  *          ⇒   *        ⇒   f(5) =  *
        **             **                   **
        ***            ***                  ***
        ****           ****                 ****
                       *****                *****
```

$f(4)$ 가 이미 그려져 있다면 가장 아래쪽에 '*'을 5개만 그리면 바로 $f(5)$ 의 패턴이 된다. 이는 다른 패턴들 간에도 성립된다.

따라서 n 으로 일반화하면 다음과 같은 관계식을 얻을 수 있다.

$f(n)$ = $f(n-1)$, '*'을 n 개 출력한 후 [줄바꿈]

위와 같이 정의할 수 있다. 위 정의에 의하면 모든 $f(k)$ 는 항상 마지막에 [줄바꿈]으로 끝나므로 다음에 '*'를 추가할 때, 따로 줄을 바꾸지 않아도 됨을 알 수 있다.

다음으로 직접 그려야할 패턴이 필요하다. $f(n)$은 $f(n-1)$로부터 구하므로 $f(1)$ 또는 $f(0)$만 정의하면 된다.

$$f(0) = \text{아무것도 그리지 않음.}, \quad f(1) = * \text{ [줄바꿈]}$$

위 2개의 식을 이용하여 모든 $f(k)$를 그릴 수 있다. 위의 식들을 정리하면 다음과 같은 관계식이 된다.

$$f(n) = \begin{cases} * \text{ [줄바꿈]} & (n=1) \\ f(n-1),\ '*...*'(n\text{개})\text{[줄바꿈]} & (n>1) \end{cases}$$

위 관계식을 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>

void f(int n)
{
  if(n==1)
    printf("*\n");
  else
  {
    f(n-1);
    for(int i=0; i<n; i++)
      printf("*");
    puts("");
  }
}

int main()
{
  int n;
  scanf("%d", &n);
  f(n);
  return 0;
}
```

직접 그리는 항을 $f(0)$으로 설정하면 관계식은 다음과 같이 표현할 수 있다.

$$f(n) = f(n-1),\ '*...*'\ (n\text{개})\text{[줄바꿈]}\ (n>0)$$

위 관계식을 이용하면 소스코드를 다음과 같이 보다 더 간단하게 작성할 수 있다.

```
#include <cstdio>

void f(int n)
{
  if(n>0)
  {
    f(n-1);
    for(int i=0; i<n; i++)
      printf("*");
    puts("");
  }
}

int main()
{
  int n;
  scanf("%d", &n);
  f(n);
  return 0;
}
```

위 방법은 대부분의 n에 대해서는 처리할 수 있으나 입력 최댓값인 10,000을 처리하기에는 너무 효율이 떨어진다. 모든 '*'을 직접 하나씩 모두 그리고 있기 때문이다. 여기서 조금 더 활용하여 '*'을 빨리 그리는 방법에 대해서 생각해보자.

문자열 배열을 이용하여 '*'을 그리는 속도를 향상시키자.

한 줄씩 증가될 때마다 윗줄에서 사용한 '*'의 수보다 1씩 늘어난다. 따라서 문자열을 다음과 같이 이용하자.

① 빈 문자열을 준비한다.
② 한 줄 출력할 때마다 문자열 마지막에 '*'을 하나 추가하고 문자열 전체를 출력한다.
③ 모든 줄을 출력하지 않았으면 ②으로 이동한다.

위의 방법을 이용하면 한 줄의 문자를 찍을 때 '*'를 하나씩 출력하는 것이 아니라 문자열을 한꺼번에 출력하므로 속도는 훨씬 빨라진다.

위의 방법을 이용한 소스코드는 다음과 같다.

줄	코드	참고

```
1  #include <cstdio>
2  #define MAXN 10000
3  
4  char star[MAXN+1];
5  
6  void f(int n)
7  {
8    if(n>0)
9    {
10     f(n-1);
11     star[n]='*';
12     puts(star+1);
13   }
14 }
15 
16 int main()
17 {
18   int n;
19   scanf("%d", &n);
20   f(n);
21   return 0;
22 }
```

이 방법으로 패턴을 그리면 입력값 10,000에 대해서도 빠른 속도로 동작하므로 제한 시간 내에 충분히 문제를 해결할 수 있다.

11행이 실행되는 동안 문자열 star는 한 줄을 출력할 때마다 마지막에 1개씩 '*'문자가 추가된다.

12행에서 puts(star+1)에서 한꺼번에 문자열을 출력한다. 이는 printf()를 이용하여 한 번에 하나씩 '*'를 출력하는 것 보다 훨씬 속도가 빠르다.

참고로 puts()는 문자열의 시작 주소를 이용하는데, star에는 다음과 같이 저장되어 있다.

star	0	'*'	'*'	'*'	······	'*'	0
	0	1	2	3	······	n-1	n

따라서 시작주소는 star가 아니라 star+1이 되어야 하므로 puts(star+1)을 이용하여 출력해야 한다.

문제 3

combination(S)

${}_nC_k$는 n개 중에서 k개를 고르는 방법의 수이다.

${}_nC_k$를 구하는 일반식은 다음과 같다.

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1} = \frac{n!}{k!(n-k)!}$$

위 식은 $n!$을 이용하기 때문에 n이 커지면 overflow가 발생하여 정확한 값을 구할 수 없다.

물론 위 방법 이외에도 다양한 점화식으로도 구할 수 있다.

${}_nC_k$를 정확하게 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 n과 k가 공백으로 구분되어 입력된다.
(단, 1 <= n, k < 30)

출력

구한 답을 첫 번째 줄에 출력한다.

입력 예	출력 예
5 1	5
10 5	252

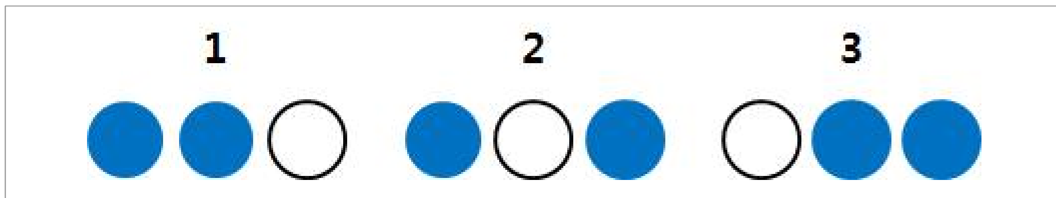
풀이

이 문제는 수학의 조합 문제이다. 문제에 주어진 공식으로도 조합의 값을 구할 수 있지만 factorial을 계산할 때 오버플로가 발생될 수 있으므로 프로그래밍으로 접근할 때는 조심해야 한다.

먼저 관계식을 세우기 위해서 상태를 정의하자.

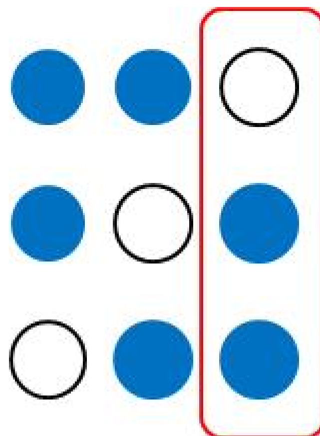
$f(n, k)$ = n개 중 k개를 고르는 경우의 수

이 문제의 경우에 n개 중 k개를 고르기 위해서 가장 마지막에 있는 n번째 경우에 집중해보자. 일단 먼저 간단한 예로 3개의 물건 중에서 2개의 물건을 고르는 모든 경우를 나열하면 다음과 같다.



위의 그림에서 파란색은 고른 경우이고 흰색은 고르지 않은 경우를 의미하며, 위의 그림과 같이 3가지가 있다.

여기서 마지막 3번째 물건만 살펴보자.



마지막 물건만 본다면 그림과 같이 고르거나, 안 고르거나 2가지 선택만 가능하다. 따라서 마지막 물건을 고르지 않는 경우의 수와 마지막 물건을 고르는 경우의 수를 따로 판단하여 구할 수 있다.

마지막 물건을 고른 경우	마지막 물건을 고르지 않은 경우

위 그림과 같이 마지막 물건을 고른 경우는 마지막 물건을 제외한 2개의 물건 중 1개를 고른 상태가 된다.

마찬가지로 마지막 물건을 고르지 않은 경우에, 앞의 2개의 물건을 관찰해 보면 2개의 물건 중 2개의 물건을 모두 고른 상태가 된다.

따라서 다음과 같은 관계를 얻을 수 있다.

$f(n, k)$ = 마지막 물건을 고른 경우의 수 + 마지막 물체를 고르지 않은 경우의 수
마지막 물건을 고른 경우의 수 = $f(n-1, k-1)$
마지막 물건을 고르지 않은 경우의 수 = $f(n-1, k)$

앞의 그림을 관계식으로 나타낸 것이다. 이를 종합하면 다음과 같은 관계식을 얻는다.

$$f(n, k) = f(n-1, k-1) + f(n-1, k)$$

그리고 직접 구해야 하는 상태는 n개의 물건 중 n개를 고르는 경우의 수는 1가지임이 명백하고, n개의 물건 중 1개를 고르는 경우의 수는 n가지임이 명백하므로, 다음과 같이 정의할 수 있다.

$$f(n,\ 1) = n, \quad f(n,\ n) = 1$$

위의 관계를 정리하여 관계식을 구하면 다음과 같다.

$$f(n,\ k) = \begin{cases} 1 & (k = n) \\ n & (k = 1) \\ f(n-1,\ k-1) + f(n-1,\ k) & (1 < k < n) \end{cases}$$

이 관계식을 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>

int f(int n, int k)
{
  if(k==n) return 1;
  else if(k==1) return n;
  else return f(n-1,k-1)+f(n-1,k);
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  printf("%d\n", f(n,k));
  return 0;
}
```

위 알고리즘은 factorial을 사용하지 않기 때문에 해가 오버플로가 나지 않는 범위라면 오버플로 없이 계산할 수 있다. 하지만 위 알고리즘은 계산량이 너무 많다. 따라서 n값이 커지면 보다 효율적인 아이디어가 필요하다.

다음의 수학식을 관계기반 설계에 이용하여 효율을 높여보자. 물론 이 경우에도 factorial은 이용하지 않도록 하는 것이 중요하다.

수학적인 $f(n, k)$의 식은 다음과 같다.

$$f(n, k) = \frac{n!}{k!\,(n-k)!}$$

다음으로 $f(n, k-1)$의 식을 유도하면 다음과 같다.

$$f(n, k-1) = \frac{n!}{(k-1)!\,(n-k+1)!}$$

위 두 식을 이용하면 다음과 같은 식을 얻을 수 있다.

$$f(n, k) = f(n, k-1) \times \frac{n-k+1}{k}$$

이 식을 관계식으로 설정하여 $f(n, k)$를 정의하면 다음과 같다.

$$f(n, k) = \begin{cases} 1 & (k = n) \\ n & (k = 1) \\ f(n, k-1) \times \dfrac{n-k+1}{k} & (1 < k < n) \end{cases}$$

이 관계식을 적용하면 효율이 매우 좋아진다. 식을 보면 선형시간임을 알 수 있다. 이 관계식을 적용한 소스코드는 다음과 같다.

줄	코드	참고
1	`#include <cstdio>`	
2		
3	`int f(int n, int k)`	
4	`{`	
5	`  if(k==n) return 1;`	
6	`  else if(k==1) return n;`	
7	`  else return f(n,k-1)*(n-k+1)/k;`	
8	`}`	
9		
10	`int main()`	
11	`{`	
12	`  int n, k;`	
13	`  scanf("%d %d", &n, &k);`	
14	`  printf("%d\n", f(n,k));`	
15	`  return 0;`	
16	`}`	

여기서도 주의할 것이 있다. 위 7행의 곱셈의 결과가 오버플로를 발생시킬 가능성을 가지고 있다. 따라서 위 식을 다음과 같이 변경해도 결과는 수학적으로 같음을 알 수 있다. 따라서 이 식을 다시 적용하자.

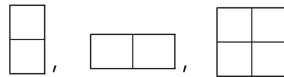
$$f(n,\ k) = \begin{cases} 1 & (k = n) \\ n & (k = 1) \\ \dfrac{n-k+1}{k} \times f(n,\ k-1) & (1 < k < n) \end{cases}$$

수학적으로는 먼저 제시한 식과 이 식이 같은 의미지만 오버플로의 발생여부는 달라진다. 따라서 이 식을 적용하는 것이 훨씬 안정적으로 동작함을 보장한다.

문제 4

타일채우기(S)

2*1 혹은 2*2크기의 타일을 2*n 크기의 직사각형모양 틀에 넣으려고 한다. 이 때 가능한 경우의 수를 구하여라.



경우의 수가 커지므로, 주어지는 수 m으로 나눈 나머지를 출력한다.

입력

첫 줄에는 직사각형 틀의 가로 길이 n이 주어진다.
둘째 줄에는 m이 주어진다. (1<=n<=100,000, 1<=m<=40,000)

출력

경우의 수를 m으로 나눈 나머지를 출력한다.

입력 예	출력 예
8 100	71
8 2	1

풀이

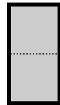
문제를 관계기반으로 해결하기 위해서는 먼저 문제의 상태를 정의해야 한다.

이러한 문제가 관계기반으로 풀리기 위해서는 타일을 채우는 순서에 관계가 없어야 한다. 이 문제는 가운데에서부터 타일을 채우나 맨 끝에서부터 채우나, 처음부터 채우나 같다. 따라서 처음부터 채워나간다고 할 때, $f(n)$을 다음과 같이 정의해보자.

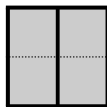
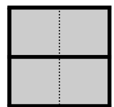
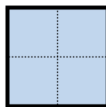
$f(n)$ = "1번 칸부터 n번째 칸까지 조건에 맞도록 채운 경우의 수"

위와 같이 정의하고, 수학적 귀납법에 의하면 반드시 직접적으로 해결해야 하는 작은 문제들이 있어야 한다. 일단 다음과 같이 작은 문제들에 대해서 직접 해를 구해 둔다. 이런 경우의 수를 세는 문제에서 아무것도 안 채우는 경우를 1가지로 카운팅해야 한다는 점을 잘 알아두어야 한다. 작은 케이스의 경우의 해는 다음과 같다.

$f(0) = 1$ "즉 아무것도 안 채우는 경우는 1가지"
$f(1) = 1$ "2×1의 판을 채우는 경우도 1가지뿐이다."

 를  와 같이 채움.

$f(2) = 3$ "2×2의 판을 채우는 경우는 3가지가 있다."

, ,  와 같이 채우는 3가지가 있다.

다음으로 관계를 정의해야 한다. 관계를 어떻게 정의하느냐에 따라 다양한 해결방법이 있을 수 있다.

먼저 첫 번째 방법을 알아보자. 문제에서 구하고자 하는 해는 $f(n)$이므로 먼저 $f(n)$을 구하기 위해 $f(1)$, $f(2)$, $\cdots$, $f(n-1)$은 이미 구해두었다고 가정한다.

어떤 $f(k)$로부터 $f(n)$을 만드는 데 마지막으로 사용할 수 있는 타일은 3가지가 있다. 먼저 2×1타일을 이용하여 $f(n)$으로 만드는 경우를 살펴보자.

이 경우 $f(n-1)$ 까지는 모두 채워져 있어야 2×1 타일을 이용하여 $f(n)$을 만들 수 있다.

$n-1$

위 그림과 같이 $n-1$ 칸을 채운 경우의 수를 $f(n-1)$ 이라고 알고 있다고 가정하자. 이 상태에서 $f(n)$을 구하기 위하여 2×1 의 타일을 이용할 수 있다.

n

이와 같이 $f(n)$을 만드는 경우의 수는 얼마일까?

위의 그림과 같이 $n-1$ 가지를 채운 경우로부터 2×1 의 타일을 이용하여 $f(n)$ 을 만들 수 있는 경우는 그냥 뒤에 붙이는 1가지뿐이다. 따라서 $f(n)$ 중 마지막에 도미노를 세우는 방법은 $f(n-1)$이라는 사실을 알 수 있다.

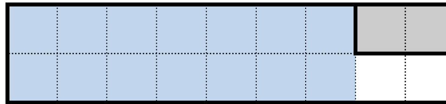
즉 2×1 타일을 이용하여 $f(n)$을 만들 경우의 수는 $f(n-1)$ 과 같다. 즉 $f(n-1)$ 을 모두 채우는 경우의 수를 알면 $f(n)$ 중 2×1 을 이용하여 채우는 경우의 수를 알 수 있다.

다음으로 1×2 타일을 이용하여 $f(n)$ 을 만드는 경우의 수를 구해보자. 이 경우 그림으로 살펴보면 적어도 $f(n-2)$ 까지는 채워진 상태를 알 수 있어야 마지막으로 1×2 를 이용하여 $f(n)$을 만들 수 있는 경우의 수를 알 수 있다.



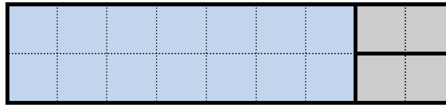
$n-2$

위 그림과 같이 $n-2$칸을 채운 경우의 수를 $f(n-2)$ 라고 알고 있다고 가정하자. 이 상태에서 $f(n)$을 구하기 위하여 1×2의 타일을 이용할 수 있다. 먼저 1개를 배치해보면 다음과 같다.



n

이 때 마지막 부분이 비어있게 된다. 이 부분을 채울 수 있는 방법은 다시 1×2타일을 하나 더 쓰는 수밖에 없다. 따라서 마지막을 1×2타일로 채우는 방법은 다음 그림과 같다.



n

위와 같은 방법 1가지뿐이다. 따라서 마지막에 1×2타일을 이용하여 $f(n)$을 만들 수 있는 방법은 $f(n-2)$가지임을 알 수 있다.

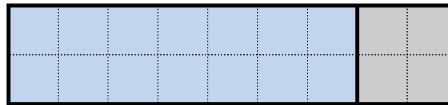
$f(n)$을 만들기 위해서 마지막으로 1×2 타일을 이용하는 경우의 수는 $f(n-2)$ 가지임을 알 수 있다.

마지막으로 2×2 타일을 이용하여 $f(n)$을 만드는 경우의 수만 구하면 된다. 이 경우도 다음 그림과 같이 쉽게 구할 수 있다.



$n-2$

위 그림과 같이 $n-2$칸을 채운 경우의 수를 $f(n-2)$라고 알고 있다고 가정하자. 이 상태에서 $f(n)$을 구하기 위하여 2×2의 타일을 이용할 수 있다. 먼저 1개를 배치해보면 다음과 같다.



n

이와 같이 $f(n)$을 만들 수 있으며, 위와 같은 방법 1가지뿐이다. 따라서 마지막에 2×2타일을 이용하여 $f(n)$을 만들 수 있는 방법은 $f(n-2)$가지임을 알 수 있다.

따라서 앞 세 가지 경우의 수를 모두 더하면 모든 $f(n)$을 만들 수 있는 경우의 수를 알 수 있다. 하지만 이 방법은 실제 값을 구하는 것이 아니라 관계를 만들어 낸 것이다. 왜냐하면 실제로 $f(n-1)$, $f(n-2)$ 등의 값을 구한 것이 아니라 미리 구해두었다고 가정했기 때문이다.

따라서 위 관계로부터 각 사건은 합의 법칙이 적용됨을 알 수 있으며, 이를 통하여 얻어낸 관계식 혹은 점화식은 다음과 같다.

$$
\begin{aligned}
f(n) &= f(n-1) + f(n-2) + f(n-2) \\
&= f(n-1) + 2f(n-2)
\end{aligned}
$$

위 관계식을 이용하려면 적어도 2개의 초깃값이 필요하다. 이미 초깃값들은 구해 두었으므로 다음과 같이 정리할 수 있다.

$$f(n) = \begin{cases} 1 & (n=0) \\ 1 & (n=1) \\ f(n-1) + 2f(n-2) & (n>1) \end{cases}$$

혹은

$$f(n) = \begin{cases} 1 & (n=1) \\ 3 & (n=2) \\ f(n-1) + 2f(n-2) & (n>2) \end{cases}$$

위 두 식은 같은 의미로 볼 수 있다. 위 두 가지 관계식 중 첫 번째 관계식은 다음과 같이 고쳐도 무방하다.

$$f(n) = \begin{cases} 1 & (n \leq 1) \\ f(n-1) + 2f(n-2) & (n>1) \end{cases}$$

이 관계식을 재귀함수를 이용하여 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <cstdio>

int n, m;

int f(int k)
{
    if(k<=1) return 1%m;
    else return (f(k-1)+2*f(k-2))%m;
}

int main()
{
    scanf("%d %d", &n, &m);
    printf("%d\n", f(n));
    return 0;
}
```

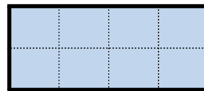
사실 이 문제에서 구하는 해는 값이 너무 크므로 m으로 나눈 나머지를 구하기 때문에 소스코드에 이 부분을 추가했다. 여기서 주의할 점은 7행의 직접 구하는 값에도 m으로 나누어 줘야한다는 점이다.

이 방법은 쉽게 구현은 가능하나, 계산량이 너무 많아서 n이 40정도의 값이더라도 너무 많은 시간이 걸린다. 대략적인 계산량은 $O(2^n)$정도이다. 따라서 이 경우는 일부 부분점수만 획득할 수 있다.

이 문제를 보다 효율적으로 처리하기 위해서는 새로운 관계식을 유도할 필요가 있다. 새로운 아이디어를 만들어 보자.

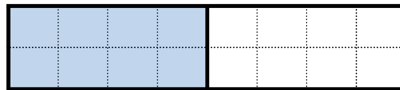
앞에서 1부터 n까지의 합을 구할 때 $f(k)$를 $f(\frac{n}{2})$로 설정하여 효율을 높였었다. 이 문제에서도 이 원리가 적용가능하다.

먼저 다음과 같이 접근해보자. $f(\frac{n}{2})$을 미리 구해두었다고 가정하면 다음과 같은 관계식을 얻을 수 있다.



$\frac{n}{2}$

위 그림과 같이 $\frac{n}{2}$칸을 채운 경우의 수를 $f(\frac{n}{2})$이라고 알고 있다고 가정하자. 이 상태에서 $f(n)$을 구하기 위하여 어떻게 해야 할까?



n

양쪽의 모양을 보자.
양쪽의 모양이 같다. 즉 모양이 같다면 채우는 방법도 같다는 의미이다. 왼쪽 부분을 채우는데 $f(\frac{n}{2})$가지임을 이미 알고 있다.

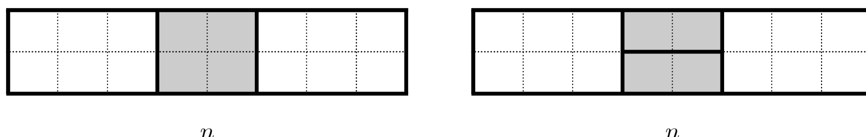
그렇다면 전체를 채우는 방법은 $f(\frac{n}{2}) \times f(\frac{n}{2})$라는 사실을 알 수 있다. 이는 곱의 법칙을 이용한 것이다.

앞의 방법으로 모든 경우의 수를 구할 수 있는가? 아니다. 앞 경우는 n이 짝수일 경우, 정확하게 반으로 나눌 수 있을 때의 경우의 수다.

타일을 채운 방법들 중 가운데를 반으로 나눌 수 있도록 채울 수도 있지만, 가운데를 반으로 나눌 수 없도록 채우는 경우도 있다.

즉, 위에서 구한 방법은 칸의 수 n이 짝수이고, 가운데를 반으로 나눌 수 있도록 채웠을 때의 경우의 수를 구한 것이다.

다음으로 가운데를 반으로 나눌 수 없는 경우에 대해서 생각해보자. 가운데를 반으로 나눌 수 없도록 채우는 방법은 다음 2가지뿐이다.



위의 2가지의 경우 흰 영역을 채우는 문제가 된다.

이 때 채우는 경우의 수는 왼쪽의 부분이 $f(\frac{n}{2}-1)$ 가지이고, 오른쪽과 왼쪽의 모양이 같으므로 $f(\frac{n}{2}-1) \times f(\frac{n}{2}-1)$이고 가운데를 나눌 수 없는 경우가 위의 그림과 같이 2가지이므로 결론은 다음과 같다.

$$2 \times f(\frac{n}{2}-1) \times f(\frac{n}{2}-1)$$

앞 2가지의 경우를 종합하면 n이 짝수인 경우는 모두 구할 수 있다. 그렇다면 홀수인 경우는 어떻게 처리할까?

조금 생각해보면 홀수일 경우에는, 먼저 구했던 관계식 $f(n)$ $f(n-1)+2f(n-2)$를 그대로 이용할 수 있다. 이를 적용시킨 관계식은 다음과 같다.

$$f(n) = \begin{cases} 1 & (n \le 1) \\ f(\frac{n}{2})^2 + 2f(\frac{n}{2}-1)^2 & (n > 1,\ n\text{은 짝수}) \\ f(n-1) + 2f(n-2) & (n > 1,\ n\text{은 홀수}) \end{cases}$$

위 점화식으로 구현한 소스코드는 다음과 같다.

줄 | 코드 | 참고

```
#include <cstdio>

int n, m;

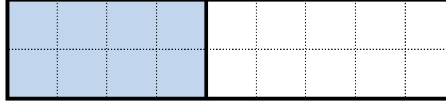
int f(int k)
{
  if(k<=1) return 1%m;
  else if(k%2) return (f(k-1)+2*f(k-2))%m;
  else return (f(k/2)*f(k/2)+2*f(k/2-1)*f(k/2-1))%m;
}

int main()
{
  scanf("%d %d", &n, &m);
  printf("%d\n", f(n));
  return 0;
}
```

이 방법은 앞의 방법에 비해 효율이 매우 좋다. 하지만 여전히 n이 100,000을 해결할 정도는 아니다. 하지만 위 방법은 매우 훌륭한 아이디어를 포함하고 있으므로 꼭 익혀둘 수 있도록 한다.

마지막으로 홀수일 때도 $\frac{n}{2}$을 이용하여 구하는 관계식을 유도해보자. 먼저 홀수일 경우에는 정확하게 반으로 나눌 수 없으므로 $\left[\frac{n}{2}\right]$의 값을 이용하여 나눠질 경우와 그렇지 않을 경우를 그림으로 나타내면 다음과 같다.

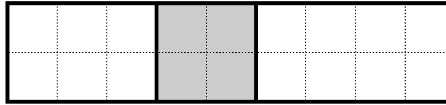
n이 홀수 일 때, 다음 그림과 같이 2가지로 나뉘는 경우를 채워보자.



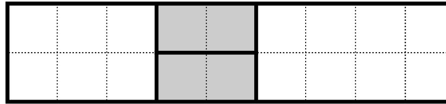
n

위의 경우와 같은 부분을 정확히 나눌 수 있도록 전체를 채우는 방법은 $f(\left[\frac{n}{2}\right])\times f(\left[\frac{n}{2}\right]+1)$ 이라는 사실을 알 수 있다.

다음으로 나눌 수 없는 경우를 알아보자.



n



n

위의 2가지의 경우 흰 영역을 채우는 문제가 된다.

이 때 채우는 경우의 수는 왼쪽의 부분이 $f(\left[\frac{n}{2}\right]-1)$ 가지이고, 오른쪽은 $f(\left[\frac{n}{2}\right])$ 가지임을 알 수 있다. 이러한 경우가 2가지이므로 관계식은 다음과 같다.

$$2\times f(\left[\frac{n}{2}\right]-1)\times f(\left[\frac{n}{2}\right])$$

따라서 다음과 같은 관계식을 얻을 수 있다.

$$f(n)=\begin{cases} 1 & (n\le 1) \\ f(\frac{n}{2})^2+2f(\frac{n}{2}-1)^2 & (n>1,\ n\text{은 짝수}) \\ f(\left[\frac{n}{2}\right])f(\left[\frac{n}{2}\right]+1)+2f(\left[\frac{n}{2}\right]-1)f(\left[\frac{n}{2}\right]) & (n>1,\ n\text{은 홀수}) \end{cases}$$

이 관계식을 이용하여 구현한 소스코드는 다음과 같다. 위 식에서 n이 홀수인 경우에 이용한 가우스함수처리는 정수연산에서는 나누는 연산이 자동으로 해결해주므로 따로 처리할 필요는 없다.

코드	참고

```
#include <cstdio>

int n, m;

long long int f(int k)
{
  if(k<=1) return 1%m;
  else if(k%2) return(f(k/2)*f(k/2+1)+2*f(k/2)*f(k/2-1))%m;
  else return (f(k/2)*f(k/2)+2*f(k/2-1)*f(k/2-1))%m;
}

int main()
{
  scanf("%d %d", &n, &m);
  printf("%lld\n", f(n));
  return 0;
}
```

관계식에 제곱 등의 값이 들어가므로 오버플로에 주의해야 한다. 따라서 함수값을 long long int로 바꾸어 처리했다.

이 알고리즘의 경우에는 충분히 제한 시간 이내에 통과할 수 있다.

이와 같이 같은 관계기반 설계라도 관계를 설계하는 방법에 따라 효율이 매우 달라질 수 있다. 관계를 설계할 때 보다 효율적으로 설계할 수 있도록 연습하자. 이러한 기법을 분할 정복이라고 하는데, 먼저 다루었던 관계식 $f(n-1)+2f(n-2)$는 사례를 한 번에 정복할 수 있는 사례와 그렇지 않은 사례로 나눈 것으로 이러한 분할을 비균등 분할이라고 한다.

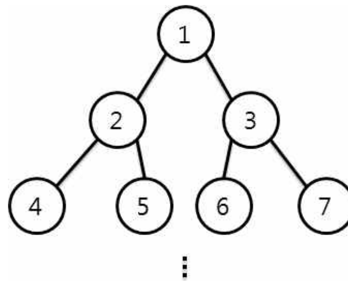
분할정복법으로 문제를 해결할 때 가장 효율적인 방법은 같은 크기의 분할을 k개로 하는 것이 가장 이득이다. 특히 k를 2로 할 때 설계 및 구현이 쉬운 경우가 많다. 이 해법의 경우 k를 2로 균등 분할한 알고리즘이므로 매우 빠른 시간에 해를 구할 수 있다.

이 방법을 이용하여 피보나치 수를 매우 효율적으로 구할 수 있는 알고리즘도 작성할 수 있으니 참고하기 바란다.

문제 5

distance of nodes

다음과 같이 노드에 번호가 부여된 이진트리가 있다. 이 트리는 단말 노드를 제외한 모든 노드들이 2개의 자식을 가지며, 모든 단말노드의 높이가 같다.



두 노드 간의 거리는 한 노드에서 다른 노드로 이동할 때 거치는 간선의 수의 합을 의미한다. 예를 들어 노드 5와 노드 3의 거리는 3이다.

임의의 두 노드가 주어질 때, 두 노드 간의 거리를 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 두 노드의 번호가 공백으로 구분되어 입력된다.
(단, 두 노드의 값은 1이상 2,100,000,000 이하의 값이다.)

출력

두 노드 간의 거리를 출력한다.

입력 예	출력 예
5 3	3
3 4	3

풀이

이 문제는 트리에서 임의의 두 노드가 주어질 때, 두 노드간의 거리를 구하는 것이다. 만약 그래프 문제라면 여러 개의 가능한 경로들 중 가장 짧은 것을 찾는 문제가 되겠지만, 이 경우는 트리 문제이므로 두 노드 간의 경로는 오직 하나만 존재한다.

물론 깊이우선탐색, 너비우선탐색 등의 다양한 방법으로 도전할 수 있지만 이 문제의 노드 수가 21억 개나 되므로 일반적인 방법으로는 제한된 시간 내에 해결할 방법이 없다. 물론 수학적으로 일반식을 유도할 수 있지만 이 단원에서는 관계기반 설계법으로 해결해 본다.

먼저 두 노드 간의 거리를 구하기 위한 관계식을 유도해야 한다. 이 문제는 완전이진트리의 형태로 주어지며, 노드의 번호가 완전이진트리를 배열에 저장할 때의 인덱스와 같은 형태로 주어지기 때문에 계산하기가 쉬운 문제가 된다.

이러한 형태의 트리에서 현재 노드의 번호를 c라고 할 때, 이 노드의 부모, 왼쪽자식, 오른쪽자식 간에는 다음과 같은 관계가 성립한다.

c노드의 부모노드 = $\left\lfloor \frac{c}{2} \right\rfloor$, c노드의 왼쪽자식 = $2c$, c노드의 오른쪽자식 = $2c+1$

이 관계를 먼저 이해하면 이 문제에서 노드의 번호가 클수록 루트와의 거리가 더 멀리 있음을 알 수 있다.

이 문제에서 두 노드의 거리를 구하기 위해서 반드시 거쳐야할 절차가 두 노드의 공통조상노드 중 가능 루트와의 거리가 가장 먼 조상노드를 찾아야 한다. 이렇게 트리에서 두 노드의 공통조상 중 가장 가까운 조상 노드를 구하는 문제를 LCA(lowest common ancestor)문제라고도 한다.

문제를 해결하는 기본 과정은 다음과 같다.

① 두 노드의 LCA를 찾는다.
② LCA로부터 각 노드의 거리를 구한다.
③ 구한 두 노드의 거리의 합이 두 노드간의 거리가 된다.

이 문제를 해결하기 위하여 먼저 상태를 함수로 정의한다.

$$f(a,\ b)\ =\ \text{두 노드 } a\text{와 } b\text{ 와의 거리}$$

이와 같이 상태를 정의했을 때, $a = b$ 라면 같은 노드이므로 거리가 0이 된다. 따라서 직접 구하는 초기 상태는 이 값으로 설정하면 된다.

$$f(a,\ b)\ =\ 0\quad (\text{단},\ a = b)$$

다음으로 상태들 간의 관계를 생각해보자. 두 노드의 조상을 찾기 위해서 두 노드가 가장 먼저 만날 수 있는 지점을 찾는 것이 중요하다.

이 문제에서 주어진 정보에 의하면 두 값 a와 b중 더 큰 값이 루트로부터 더 멀리 떨어져 있으므로 더 먼 노드를 한 단계 위로 올리면 다시 새로운 상태가 되고 이 때 이동한 거리는 1이 된다.

이 과정을 반복하면 두 노드의 LCA를 찾을 수 있다. 이 과정을 관계식으로 나타내면 다음과 같다.

$$f(a,\ b)\ =\ f(a,\ \frac{b}{2})\ +\ 1\ \ (b > a)$$
$$f(a,\ b)\ =\ f(\frac{a}{2},\ b)\ +\ 1\ \ (a > b)$$

위 식은 두 노드 중 루트로부터 더 먼 노드를 한 단계씩 이동시키는 과정을 나타낸 것이다. 이 때 이동한 거리가 1이므로 1을 더하게 된다. 이 과정을 반복하다보면 어느 순간 두 노드의 공통조상노드를 만나게 된다.

이 식들을 정리한 결과는 다음과 같다.

$$f(a,\ b)\ =\ \begin{cases} 0 & (a = b) \\ f(a,\ \frac{b}{2}) + 1 & (b > a) \\ f(\frac{a}{2},\ b) + 1 & (a > b) \end{cases}$$

이 관계식을 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>

int f(int a, int b)
{
  if(a==b) return 0;
  if(b>a) return f(a,b/2)+1;
  if(a>b) return f(a/2,b)+1;
}

int main()
{
  int a, b;
  scanf("%d %d", &a, &b);
  printf("%d\n", f(a,b));
  return 0;
}
```

문제 6

영역 구분

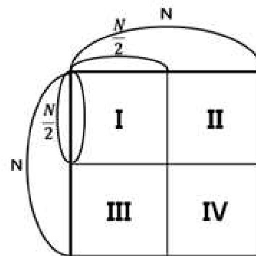
아래 그림과 같이 여러 개의 정사각형 칸들로 이루어진 정사각형 모양의 영역이 주어져 있고, 각 정사각형 칸들은 정올이의 땅은 흰색으로 칠해져 있고 영재의 땅은 검은색으로 칠해져 있다. 주어진 땅을 일정한 규칙에 따라 나누어 다양한 크기를 가진 정사각형 모양의 하얀색 또는 회색 영역으로 구분하려고 한다.

1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
1	0	0	0	1	1	1	1
0	1	0	0	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1

[그림 1]

전체 영역의 크기가 n*n (n= 2^k, k는 1 이상 7 이하의 자연수) 이라면 영역을 구분하는 규칙은 다음과 같다. 전체 영역이 모두 같은 색이 아니라면 가로와 세로로 중간 부분을 잘라서 <그림 2>의 Ⅰ, Ⅱ, Ⅲ, Ⅳ와 같이 똑같은 크기의 네 개의 n/2 * n/2 영역으로 나눈다.

나누어진 영역 Ⅰ, Ⅱ, Ⅲ, Ⅳ 각각에 대해서도 앞에서와 마찬가지로 모두 같은 색으로 이루어지지 않으면 같은 방법으로 똑같은 크기의 네 개의 영역으로 나눈다. 이와 같은 과정을 구분되어진 영역이 모두 하얀색 또는 모두 회색으로 되거나, 하나의 정사각형 칸이 되어 더 이상 나눌 수 없을 때까지 반복한다.



[그림 2]

위와 같은 규칙에 따라 나누었을 때 [그림 3]은 [그림 1]의 영역을 처음 나눈 후의 상태를, [그림 4]는 두 번째 나눈 후의 상태를, [그림 5]는 최종적으로 만들어진 다양한 크기의 9개의 하얀색 영역과 7개의 회색영역을 보여주고 있다.

영역 구분 (계속)

1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
1	0	0	0	1	1	1	1
0	1	0	0	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1

[그림 3]

1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
1	0	0	0	1	1	1	1
0	1	0	0	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1

[그림 4]

1	1	0	0	0	0	1	1
1	1	0	0	0	0	1	1
0	0	0	0	1	1	0	0
0	0	0	0	1	1	0	0
1	0	0	0	1	1	1	1
0	1	0	0	1	1	1	1
0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1

[그림 5]

입력으로 주어진 영역의 한 변의 길이 n과 각 정사각형 칸의 색 (하얀색 또는 회색)이 주어질 때 잘라진 하얀색 영역과 회색 영역의 개수를 구하는 프로그램을 작성하시오.

입력

입력 파일의 첫째 줄에는 전체 영역의 한 변의 길이 n이 주어져 있다. n은 2, 4, 8, 16, 32, 64, 128 중 하나이다. 영역의 각 가로줄의 정사각형 칸들의 색이 윗 줄부터 차례로 입력 파일의 둘째 줄부터 마지막 줄까지 주어진다. 회색으로 칠해진 칸은 0, 하얀색으로 칠해진 칸은 1로 주어지며 각 숫자 사이에는 빈칸이 하나씩 있다.

출력

첫째 줄에는 잘라진 하얀색 영역의 수를 출력하고 둘째 줄 에는 회색 영역의 수를 출력한다.

입력 예	출력 예
8 1 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 0 0 0 1 1 1 1 0 1 0 0 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1	9 7

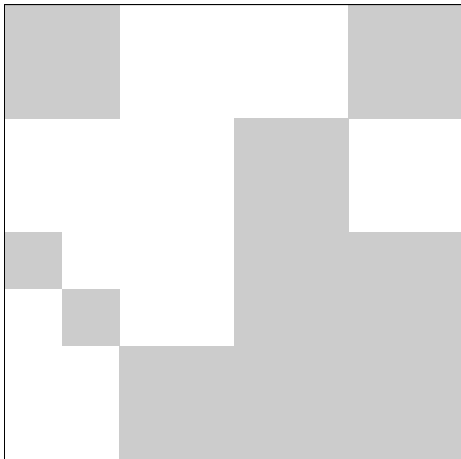
출처: 한국정보올림피아드(2001 전국본선 중등부)

풀이

이 문제는 맵 전체를 루트로 보고 각 사분면을 기준으로 4등분하여 어떤 조건을 만족할 때까지 반복적으로 분할해 나간다.

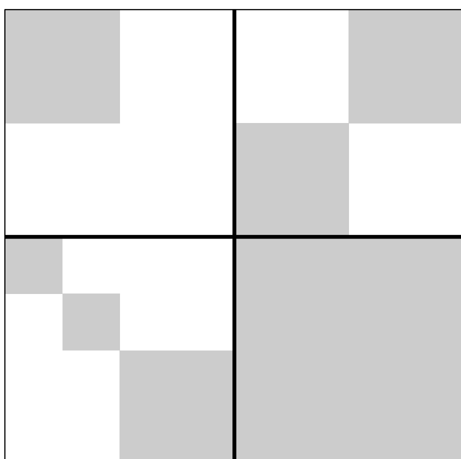
일단 입력으로 들어오는 모든 케이스는 변의 길이가 2^k꼴이기 때문에 정확하게 분할하여 처리할 수 있다.

먼저 입력 예시를 처리하는 과정을 살펴보면 다음과 같다.



한 변의 길이가 8인 입력형태의 종이를 나타낸다. 모든 종이가 같은 모든 종이가 같은 색깔이 아니므로 4 등분으로 분할한다.

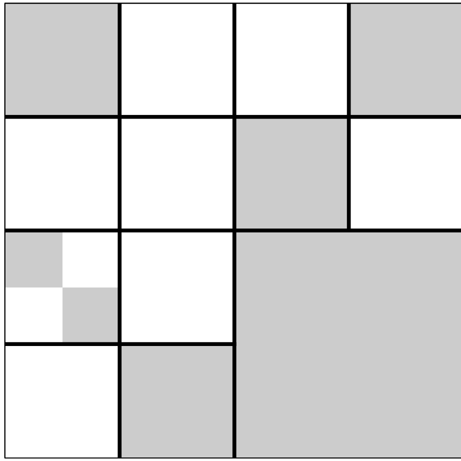
혼합색종이 = 1
하얀색종이 = 0
회색종이 = 0



한 번 분할된 한 번의 길이가 4인 종이인 상태.

좌표평면상에서의 사분면으로 볼 때, 4 사분면을 제외하면 모두 색깔이 한 가지가 아니므로 다시 길이가 2인 종이로 분할.

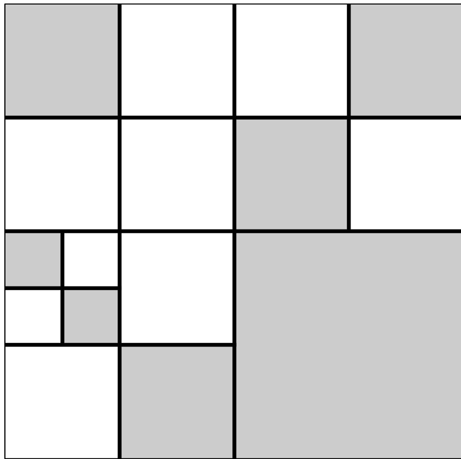
혼합색종이 = 3
하얀색종이 = 0
회색종이 = 1



길이가 2인 사각형들 중 3사분면 중 2사분면의 면을 제외하고는 모두 한 가지 색깔의 종이가 되었다.

따라서 3사분면 중 2사분면 만 다시 길이가 1인 종이로 분할한다.

혼합색종이 = 1
하얀색종이 = 7
회색종이 = 5



이제 더 이상 다른 색깔이 섞인 종이는 없이 모두 한 가지 색깔의 종이로 분할 되었다.

각 색깔의 결과는 다음과 같다.

혼합색종이 = 0
하얀색종이 = 9
회색종이 = 7

이와 같은 단계로 처리하기 위해 다음과 같이 상태를 정의한다.

$f(a, b, n)$ = a행 b열을 왼쪽, 위 모서리로 하고 변의 길이가 n인 종이를 탐색

이 상태를 다음과 같은 관계식으로 정의할 수 있다.

$$f(a,\ b,\ n) = \begin{cases} 1 & (n=1,\ S[a][b]=1)\ \text{또는}\ (S[a][b] \sim S[a+n-1][b+n-1]\ \text{까지 모두 } 1) \\ 0 & (n=1,\ S[a][b]=0)\ \text{또는}\ (S[a][b] \sim S[a+n-1][b+n-1]\ \text{까지 모두 } 0) \\ f(a,\ b,\ \frac{n}{2}),\ f(a+\frac{n}{2},\ b,\ \frac{n}{2}), \\ f(a,\ b+\frac{n}{2},\ \frac{n}{2}),\ f(a+\frac{n}{2},\ b+\frac{n}{2}) & (\text{그 외}) \end{cases}$$

위의 관계식에 따라 작성한 소스코드는 다음과 같다.

```
#include <stdio.h>
int s[1<<7][1<<7], n, gray, white;

void f(int a, int b, int n)
{
  bool isOne=1;
  for(int i=a; i<a+n; i++)
    for(int j=b; j<b+n; j++)
      if(s[a][b]!=s[i][j])
        isOne=0;
  if(isOne)
  {
    if(s[a][b]==1) white++;
    else gray++;
    return;
  }
  else
  {
    f(a, b, n/2);
    f(a+n/2, b, n/2);
    f(a, b+n/2, n/2);
    f(a+n/2, b+n/2, n/2);
  }
}

int main()
{
```

줄	코드	참고
28	`  int i,j;`	
29	`  scanf("%d", &n);`	
30		
31	`  for(i=0; i<n; i++)`	
32	`    for(j=0; j<n; j++)`	
33	`      scanf("%d", &s[i][j]);`	
34	`  f(0,0,n);`	
35	`  printf("%d\n%d\n", gray, white);`	
36	`  return 0;`	
37	`}`	

위 소스코드에서 2행에 1<<7 이란 표현을 사용했는데 이는 2^7과 같은 의미이다. 즉 1을 왼쪽으로 7번 시프트 했으므로 이진수로 $10000000_{(2)}$이 되므로 128로 설정하는 것이다. 2^k 꼴을 표현할 때 시프트연산을 이용하면 편리하다.

위 함수를 수학식 함수로 정의하여 사용할 수도 있다.

$f(a,\ b,\ n,\ 0)$ = a행 b열을 왼쪽, 위 모서리로 하고 변의 길이가 n인 종이를 구성하는 하얀색 종이의 수

$f(a,\ b,\ n,\ 1)$ = a행 b열을 왼쪽, 위 모서리로 하고 변의 길이가 n인 종이를 구성하는 회색 종이의 수

위 식을 이용하여 하얀색 종이의 수를 구하는 관계식은 다음과 같다.

$$f(a,\ b,\ n,\ 0) = \begin{cases} 0 & (n=1,\ c=0)\text{ 또는 }(\text{전 영역이 1이며, } c=0) \\ f(a,\ b,\ \frac{n}{2},0)+f(a+\frac{n}{2},\ b,\ \frac{n}{2},\ 0)+ \\ f(a,\ b+\frac{n}{2},\ \frac{n}{2},\ 0)+f(a+\frac{n}{2},\ b+\frac{n}{2},\ 0) & (\text{그 외}) \end{cases}$$

위 식을 이용하여 회색 종이의 수를 구하는 관계식은 다음과 같다.

$$f(a,\ b,\ n,\ 1)\ =\begin{cases} 1 & (n=1,\ c=1)\text{ 또는 }(\text{전 영역이 }1\text{이며, }c=1) \\ f(a,\ b,\ \frac{n}{2},1)+f(a+\frac{n}{2},\ b,\ \frac{n}{2},\ 1)+ \\ f(a,\ b+\frac{n}{2},\ \frac{n}{2},\ 1)+f(a+\frac{n}{2},\ b+\frac{n}{2},\ 1) & (\text{그 외}) \end{cases}$$

위 관계식들을 이용하여 구현한 소스코드는 다음과 같다.

줄	코드	참고

```
#include <cstdio>

int P[1<<7][1<<7], n;

bool isOne(int a, int b, int s, int v)
{
  for(int i=a; i<a+s; i++)
    for(int j=b; j<b+s; j++)
      if(P[i][j]!=v) return false;
  return true;
}

int f(int a, int b, int s, int v)
{
  if(s==1) return P[a][b]==v;
  if(isOne(a,b,s,v)) return 1;
  return f(a,b,s/2,v)+f(a+s/2,b,s/2,v)+f(a,b+s/2,s/2,v)+f(a+s/2,b+s/2,s/2,v);
}

int main()
{
  scanf("%d",&n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &P[i][j]);
  printf("%d\n%d\n", f(0,0,n,0), f(0,0,n,1));
}
```

문제 7

이진 암호화

이진 압축이란 {0, 1}로 이루어진 길이가 2^k인 문자열에 대해서, 모두 같은 문자가 될 때까지 크기가 2^{k-1}인 두 그룹으로 분할하여 모두 같은 문자가 되도록 하는 과정으로 암호화를 한다. 만약 주어진 원문이 길이가 4인 "0000"라면, 암호문은 "0"이다.

만약 주어진 암호문이 길이가 4인 "1101"이라면 모든 문자가 같지 않기 때문에 "11 01"로 일단 한 번 분할하고 다시 뒷부분은 "01"로 같지 않으니 "11 0 1"로 분할한다.

즉 이로써 만들어진 암호문은 "-1-01"이 된다. "-1-01" 의 의미는

- : 먼저 전체를 분할하고
1 : 분할된 왼쪽 부분은 모두 1이고
- : 오른쪽 분할은 다시 분할하며,
01 : 분할된 결과는 0과 1이라는 의미이다.

길이가 n인 원문을 입력받아서 암호문을 출력하는 프로그램을 작성하시오.

입력

첫 번째 줄에 암호문의 길이 n이 입력된다.
두 번째 줄에 길이가 n인 0, 1로 구성된 암호문이 입력된다.
(단, 1 <= n <= 2^{18}(단, n은 모두 2^k(k는 자연수))

출력

위 원리로 만들어진 암호문을 출력한다.

입력 예	출력 예
4 0000	0
4 1101	-1-01

풀이

이 문제는 앞에서 다루었던 영역구분에서 활용한 쿼드트리를 이진트리로 단순화한 문제라고 할 수 있다. 따라서 기본적인 처리방법은 유사하다.

길이가 8인 다음과 같은 입력 자료가 주어졌을 때, 처리하는 과정을 통하여 아이디어를 만들어 보자.

0	0	1	1	1	1	0	1

입력으로 길이가 8인 문자열이 입력된다. 처음에 모든 영역이 같은 값이 아니므로 반으로 분할한다.

현재 압축 문자열 : -

0	0	1	1	1	1	0	1

길이가 4인 2개의 문자열로 분할된 상태이다. 분할된 2개의 영역 모두 같은 값으로 구성되어 있지 않으므로 왼쪽영역부터 분할한다.

현재 압축 문자열 : --

0	0	1	1	1	1	0	1

왼쪽의 길이가 2인 2개의 영역은 모두 같은 값으로 채워져 있으므로 더 이상 분할할 필요가 없다.

이제 남은 오른쪽 길이가 4인 영역을 다시 2영역으로 분할한다.

현재 압축 문자열 : --01-

0	0	1	1	1	1	0	1

길이가 2인 영역 중 왼쪽은 모두 1로 더 이상 분할할 필요가 없으며, 마지막으로 오른쪽 길이가 2인 영역을 분할한다.

현재 압축 문자열 : --01-1-

0	0	1	1	1	1	0	1

모든 영역이 하나의 숫자로 구성된 부분으로 분할이 완료되었다. 압축 문자열은 다음과 같다.

현재 압축 문자열 : --01-1-01

이와 같은 과정으로 처리하는 상태를 정의하자.

$f(a,\ n)\ =\ S[a]$로부터 시작하여 길이가 n인 영역을 암호문자열로 치환한 결과

직접 암호화를 구현할 수 있는 정의는 다음과 같다.

$f(a,\ n)\ =\ 0\ (n=1,\ S[a]=0)$ 또는 (S의 영역이 모두 0)
$f(a,\ n)\ =\ 1\ (n=1,\ S[a]=1)$ 또는 (S의 영역이 모두 1)

위 식들을 정리하면 다음과 같은 관계식을 만들 수 있다.

$$f(a,\ n)\ =\begin{cases} 0 & (n=1,\ S[a]=0)\ \text{또는}\ (S\text{의 전영역의 값이 }0) \\ 1 & (n=1,\ S[a]=1)\ \text{또는}\ (S\text{의 전영역의 값이 }1) \\ f(a,\ \frac{n}{2}),\ f(a+\frac{n}{2},\ \frac{n}{2}) & (\text{그 외}) \end{cases}$$

위 관계식을 이용하여 구현한 소스코드는 다음과 같다.

줄	코드	참고

```
1  #include <cstdio>
2
3  char S[1<<19];
4  int n;
5
6  void f(int k, int s)
7  {
8    int sum=0;
9    if(s==1)
10   {
11     printf("%c", S[k]);
12     return;
13   }
14   for(int i=k; i<k+s; i++)
15     sum += (S[i]-'0');
16   if(sum ==0 sum==s)
17     printf("%d", (bool)sum);
18   else
19   {
20     printf("-");
21     f(k, s/2);
22     f(k+s/2, s/2);
23   }
24 }
25
26 int main()
27 {
28   scanf("%d %s", &n, S);
29   f(0,n);
30   return 0;
31 }
```

참고:
3: 524288
17: 모든 영역이 같은 숫자면 0또는 1을 출력
19: 분할

문제 8

이진 복원

이진 암호의 복원은 앞의 문제 [이진 암호화]로 만들어진 문자열을 복원하는 것이다.

이진 암호로 만들어진 암호문으로부터 원문을 복원하는 프로그램을 작성하시오.

예를 들어 4자로 구성되었던 암호문이 다음과 같다면

-1-01

원문은

1101

로 복원해야 한다.

입력

첫 번째 줄에 원문의 문자열의 길이 n이 입력된다.
두 번째 줄에 암호문이 입력된다.

(단, 1 <= n <= 2^18, 암호문의 길이는 1,000,000자를 넘지 않는다.)

출력

복원된 원문을 출력한다.

입력 예	출력 예
4 -1-01	1101
8 1	11111111

풀이

이 문제는 앞에서 다루었던 이진암호화로 만들어진 암호문을 원문으로 복원하는 프로그램이라고 생각하면 된다.

먼저 복원하는 과정에 대해서 알아보자.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

먼저 입력된 암호문의 첫 번째 문자를 읽는다.

이 때 "-"는 분할한다는 의미이고, "0", "1"은 각 영역이 모두 "0"혹은 "1"로 이루어졌다는 의미이다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

첫 번째 문자가 "-"이므로 처음 8개의 문자가 모두 같지 않다는 의미이다. 따라서 왼쪽과 오른쪽 영역으로 나뉜다.

영역을 분할했을 때 탐색순서는 왼쪽이 우선순위가 높으므로 먼저 왼쪽 4곳에 대한 값 복원이 시작된다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

두 번째 문자도 "-"이므로 다시 두 영역으로 분할한다.

다시 왼쪽 두 칸의 영역을 탐색하는 순서가 된다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

0	0						

다음으로 읽어오는 값이 "0"이다. 따라서 현재 영역의 전체값이 0이라는 의미이므로 현재 영역의 2칸을 0으로 채운다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

0	0	1	1				

다음으로 읽어오는 값이 "1"이다.

따라서 다음의 2칸을 1로 채운다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

0	0	1	1				

그 다음 문자가 "-"이므로 다시 두 영역으로 분할한다.

다시 왼쪽 두 칸의 영역을 탐색하는 순서가 된다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

0	0	1	1	1	1		

다음으로 읽어오는 문자는 "1"이다.

따라서 현재 영역 2칸이 모두 1로 채워진다는 의미이다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

0	0	1	1	1	1		

다음 문자는 “-”이다.

따라서 다음 영역은 2부분으로 분할 된다. 마지막 한 칸 씩으로 분할된 것을 볼 수 있다.

입력된 암호문의 길이는 8

-	-	0	1	-	1	-	0	1

복원된 원문

0	0	1	1	1	1	0	1

다음 문자는 차례로 “0”과 “1”이다.

따라서 마지막 영역의 각 1칸씩은 차례로 0, 1로 채우면 된다.

이제 모든 암호문을 읽었으므로 복원된 원문은

00111101

임을 알 수 있다.

이 과정을 구현하기 위하여 함수를 다음과 같이 정의해보자.

$f(k,\ n,\ v)$ = 암호문자 v에 대해

$S[k]$로 부터 $S[k+n]$까지의 영역에 대해 암호문 복원

위 함수에서 주의할 점은 암호 문자 v는 한 상태를 진행할 때마다 다음 문자로 진행되어야 한다는 점이다. 따라서 이 알고리즘에서는 큐[2]를 이용하여 문자를 차례로 읽어오는 과정을 구현한다.

위 정의를 이용하여 다음과 같은 관계식을 유도할 수 있다.

$$f(k,\ n,\ v) = \begin{cases} \text{주어진 영역을 0으로 채움} & (v='0') \\ \text{주어진 영역을 1로 채움} & (v='1') \\ f(k,\ \frac{n}{2},\ \text{다음}\,v),\ f(k+\frac{n}{2},\ \frac{n}{2},\ \text{다음}\,v) & (v='-') \end{cases}$$

2) 큐(queue)는 먼저 입력된 자료가 먼저 출력되는 자료구조이다. (http://http://en.wikipedia.org/wiki/Queue)

위 관계식을 이용하여 구현한 소스코드는 다음과 같다.

줄 | 코드

```
1  #include <cstdio>
2  #include <queue>
3
4  std::queue<char> Q;
5  char S[1<<19];
6  int n, p;
7
8  void f(int k, int s, char v)
9  {
10   if(Q.empty())
11     return;
12   if(v=='-')
13   {
14     Q.pop();
15     f(k, s/2, Q.front());
16     Q.pop();
17     f(k+s/2, s/2, Q.front());
18   }
19   else
20   {
21     for(int i=k; i<k+s; i++)
22        S2[i]=v;
23   }
24 }
25
26 int main()
27 {
28   scanf("%d %s", &n, S);
29   for(int i=0; S[i]; i++)
30     Q.push(S[i]);
31   f(0, n, Q.front());
32   printf("%s", S2);
33 }
```

참고: 5: 524288

위 문제에서 큐를 활용하는 방법을 잘 살펴보기 바란다. 물론 큐를 이용하지 않더라도 다음과 같이 처리할 수도 있다.

줄	코드	참고

```
1  #include <cstdio>
2  
3  char S[1<<19], S2[1<<19];
4  int n, p;
5  
6  void f(int k, int s)
7  {
8    char val=S[p++];
9    if(val==NULL)
10     return;
11   if(val=='-')
12   {
13     f(k,s/2);
14     f(k+s/2,s/2);
15   }
16   else
17   {
18     for(int i=k; i<k+s; i++)
19       S2[i]=val;
20   }
21 }
22 
23 int main()
24 {
25   scanf("%d %s",&n,S);
26   f(0,n);
27   printf("%s",S2);
28 }
```

참고: 3: 524288

이와 같이 queue가 하던 역할을 S라는 배열이 처리하고 있다.

문제 9

partitioned

같은 크기의 정사각형 종이가 n장 있다.

이 종이들을 밑변을 평행하게 연결하여 몇 개를 나열했다. 맨 아래에 나열한 변의 길이는 바로 윗 변의 길이보다 같거나 길어야 한다.

예를 들어 n=5일 경우는 다음과 같은 방법으로 배치가 가능하다.



위 그림을 숫자로 다음과 같이 표현할 수 있다.

(5) (4,1) (3,2) (3,1,1) (2,2,1) (2,1,1,1) (1,1,1,1,1)

n이 입력될 때 가능한 배치를 모두 구하시오. 단 숫자형태로 표현했을 때, 사전순으로 내림차순으로 한 줄에 하나씩 출력하시오.

입력

하나의 정수 n이 입력된다. (단 n은 30미만의 값이다.)

출력

숫자형태로 한 줄에 하나씩 내림차순으로 공백으로 구분하여 출력한다.

입력 예	출력 예
5	5 4 1 3 2 3 1 1 2 2 1 2 1 1 1 1 1 1 1 1

풀이

이 문제를 분할 수 문제라고 한다. 재귀함수 설계문제로 자주 이용되는 문제이므로 잘 익혀둘 필요가 있다.

일단 이 문제를 해결하기 위하여 다음과 같은 함수를 정의하자.

$f(n, k)$ = k이하의 자연수의 합으로 n을 만들 수 있는 경우의 수

이 함수는 다음과 같은 관계가 있다.

$$f(n, k) = f(n-1, 1) + f(n-2, 2) + \cdots + f(n-k, k)$$

이 식을 간단하게 설명하면, n을 k이하의 자연수의 합으로 표한다고 할 때, 가장 마지막에 더하는 수는 1부터 k까지 k가지가 있다. 단, 이 때 사용한 수들은 오름차순으로 나열한다고 가정할 때, 이들 각각의 경우의 수를 구하는 방법은 다음과 같다.

마지막에 1을 이용해서 n을 만들 수 있는 경우의 수는 $f(n-1, 1)$이라고 할 수 있다. 그 이유는 $n-1$까지를 1이하의 합으로 표현했으므로 마지막에 1을 이용하여 n을 만들 수 있다. 마찬가지로 k까지 모두 이 방법이 성립하므로 위와 같은 식이 성립된다.

하지만 이 문제에서는 경우의 수만을 구하는 것이 아니라, 직접 그 값을 주어진 규칙에 따라 출력해야하므로 약간은 더 까다로운 문제이다.

일단 큰 수부터 출력해야 하므로 함수의 호출 순서를 n이나 k 중 작은 수로부터 1로 내려오면서 계산하도록 한다.

그리고 출력을 하기 위해서는 따로 배열에 값을 채워 한꺼번에 출력해야 한다. 다양한 기법들을 배울 수 있으므로 소스코드를 잘 분석할 수 있도록 한다.

위 아이디어로 작성한 소스코드는 다음과 같다.

```c
#include <stdio.h>

int a[30], cnt;

int min(int a, int b)
{
  return a>b ? b:a;
}

void solve(int n, int k)
{
  if(n==0)
  {
    for(int i=0; i<cnt; i++)
      printf("%d ", a[i]);
    puts("");
    return;
  }
  for(int i=min(n,k); i>=1; i--)
  {
    a[cnt++]=i;
    solve(n-i, i);
    cnt--;
  }
}

int main()
{
  int n;
  scanf("%d", &n);
  solve(n, n);
  return 0;
}
```

2 동적표(dynamic table)를 이용한 알고리즘 설계

동적표란 먼저 표의 일부를 어떤 값으로 채운 후, 나머지 칸들은 주변의 값들을 참조하여 동적으로 채워지는 표를 말한다. 다음 표를 살펴보자.

① 첫 번째 칸을 1로 채운다.

D	1									
	1	2	3	4	5	6	7	8	9	10

② i번째 칸은 $(i-1)+i$ 로 채운다. (진하게 표시된 부분)

D	1	**3**	**6**	**10**	**15**	**21**	**28**	**36**	**45**	**55**
	1	2	3	4	5	6	7	8	9	10

위 표의 내용을 보면 i번째 칸에 있는 값은 1부터 i까지 합인 것을 알 수 있다. 이 표를 이용하면 1부터 i까지의 합을 구할 수 있다.

위 표를 채울 때, $D_i = D_{i-1}+i$ 를 이용하는 방법과 $\frac{i(i+1)}{2}$를 이용하는 두 가지 방법이 있다. 전자로 표를 채우는 방식을 동적방식, 후자의 방법으로 표를 채우는 방식을 정적방식이라고 하며, 이렇게 만들어진 표를 각각 동적표, 정적표라고 한다.

정적표의 경우에는 일반식을 알아야 채울 수 있는 반면, 동적표는 일단 하나 이상의 일부 칸들만 직접 채우고 나머지는 각 값들 간의 관계만 구하면 채울 수 있기 때문에 복잡한 문제에서 동적표를 이용하여 매우 효율적인 알고리즘을 만들 수 있다.

그리고 동적표를 이용해서 얻는 가장 큰 장점은 알고리즘의 계산량을 획기적으로 줄일 수 있다는 점이다. 중급편에서 다루었던 탐색 영역의 배제 기법과는 비교할 수 없을 만큼 효과가 좋다.

다음 예제들을 통하여 효율을 알아보자.

예

n 번째 피보나치 수 $f(n) = f(n-1) + f(n-2)$ 로 나타낸다. n을 입력받아서 n 번째 피보나치 수를 구하는 프로그램을 작성하시오.(단, $n \le 100$)

풀이

주어진 점화식을 이용하여 재귀함수로 작성할 수 있다. 피보나치 수의 관계식을 다시 한 번 정리하면 다음과 같다.

$$f(n) = \begin{cases} 1 & (n \le 2) \\ f(n-1) + f(n-2) & (n > 2) \end{cases}$$

주어진 관계식을 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>

int f(int n)
{
  if(n<=2)
    return 1;
  return f(n-1)+f(n-2);
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d", f(n));
  return 0;
}
```

다음은 위 알고리즘이 걸린 시간을 측정한 것이다. 값은 40부터 측정한다.

입력값	실행시간(초)	입력값	실행시간(초)
40	0.435	48	19.163
41	0.685	49	31.064
42	1.083	50	50.210
43	1.739	51	80.838
44	2.827	52	129.340
45	4.551	53	209.532
46	7.272	:	
47	11.888	100	대략 26,241 년

이 표를 분석해 보면 n값이 1증가할 때마다 시간은 약 1.6배씩 증가한다. 이 알고리즘에서 100을 입력으로 하면 대략 26,000년의 시간이 걸린다. 이 알고리즘의 효율이 어느 정도 인지 알 수 있다.

이 알고리즘에 동적표를 적용해보자. 동적표의 정의는 다음과 같다.

$DT[n]$ = n번째 피보나치 수

동적표를 이용한 소스코드는 다음과 같다.

줄	코드	참고

```
#include <cstdio>

int DT[100001];

int f(int n)
{
  if(n<=2)
    return 1;
  if(!DT[n]) DT[n]=f(n-1)+f(n-2);
  return DT[n];
}

int main()
{
```

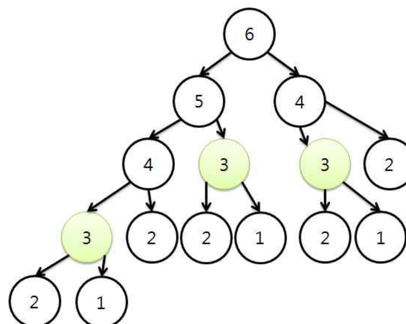
```
    int n;
    scanf("%d", &n);
    printf("%d\n", f(n));
    return 0;
}
```

단순한 재귀적 호출을 동적 테이블을 사용하는 방식으로 변경한 것뿐이다. 이 알고리즘의 효율을 동적테이블을 활용하지 않았던 알고리즘과 비교해보자.

입력값	실행시간(초)	입력값	실행시간(초)
40	0.000	48	0.000
41	0.000	49	0.000
42	0.000	50	0.000
43	0.000	51	0.000
44	0.000		
45	0.000	100	0.000
46	0.000	:	
47	0.000	50,000	0.002

두 알고리즘이 구현은 거의 같음에도 불구하고 효율의 차이는 비교할 수 없을 만큼 차이가 난다. 그 이유에 대해서 알아보자.

$n=6$일 때, 동적테이블을 활용하지 않고 재귀호출을 할 경우에는 다음과 같은 상태를 전체탐색 하게 된다.



앞의 구조에서 $f(3)$ 이 3번 호출되고 있음을 알 수 있다. 각 값이 몇 번씩 호출되는지를 정리하면 다음과 같다.

$f(6)$	$f(5)$	$f(4)$	$f(3)$	$f(2)$	…
1회	1회	2회	3회	5회	…

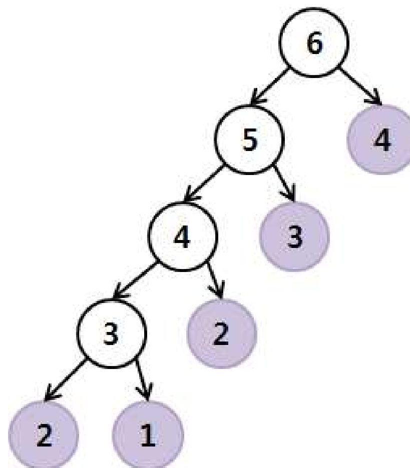
정리한 내용을 살펴보면 호출하는 횟수는 피보나치 수와 같이 증가함을 알 수 있다. 만약 $f(100)$을 호출한다면 $f(5)$는 96번째 피보나치 수만큼 호출된다.

이와 같이 재귀호출 구조는 한 번 호출한 내용을 중복해서 계속 호출하기 때문에 계산량이 폭발적으로 증가한다.

이를 방지하기 위하여 동적표를 이용하여 한 번 호출하고 그 값을 리턴할 때 동적표에 기록해두면, 나중에 다시 그 값을 호출할 필요가 있을 때에는 호출하지 않고 바로 기록한 값을 이용하여 효율을 높일 수 있다.

이렇게 중복호출을 방지하기 위하여 동적표에 데이터를 기록해두고 재사용하는 방법을 메모이제이션(memoization)이라고 한다. 이 방법은 매우 강력한 방법이므로 반드시 익힐 수 있도록 한다.

메모이제이션을 적용했을 때의 탐색구조는 다음과 같다.



위에서 색깔이 칠해진 정점들은 중복호출 없이 바로 메모한 값을 이용하여 활용하고 있다. 따라서 아래 표와 같이 모든 정점을 단 1번씩만 호출하여 해를 구할 수 있다.

$f(6)$	$f(5)$	$f(4)$	$f(3)$	$f(2)$	…
1회	1회	1회	1회	1회	…

이렇듯 동적표를 이용하면 효율이 비교할 수 없을 만큼 향상된다. 다양한 문제들을 통하여 이 기법을 익힐 수 있도록 한다.

가. 하향식 설계

하향식 설계의 기본은 위에서 아래로 내려가는 것이다. 즉, n부터 1의 방향으로 진행해 가면서 해를 구하는 설계방법을 말한다.

일반적으로 관계식(점화식)으로 설계된 경우가 많으며, $f(n)$을 호출하여 해를 구한다. 이 과정에서 필요에 따라 동적표, 즉 메모이제이션을 활용하여 효율을 높일 수 있다.

다음 알고리즘은 1부터 n까지의 합을 구하는 하향식 알고리즘이다.

코드	참고

```
#include <cstdio>

int f(int n)
{
  if(n==1) return 1;
  return f(n-1)+n;
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

이 알고리즘은 계산량이 $O(n)$이며, 1부터 n까지의 합을 하향식으로 구한 알고리즘으로 동적표는 사용하지 않은 방법이다.

```
#include <cstdio>

int f(int n)
{
  if(n==1) return 1;
  return f(n-1)+n;
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

다음은 또 다른 하향식 알고리즘이다.

```
#include <cstdio>

int f(int n)
{
  if(n==1) return 1;
  return 2*f(n/2)+((n+1)/2)*((n+1)/2);
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

이 알고리즘은 $O(\lg n)$으로 동작한다.

다음 두 알고리즘은 위의 두 알고리즘에 동적표를 적용하여 작성한 알고리즘이다.

```
#include <cstdio>

int DT[101];

int f(int n)
{
  if(n==1) return 1;
  if(!DT[n]) DT[n]=f(n-1)+n;
  return DT[n];
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

```
#include <cstdio>

int DT[101];

int f(int n)
{
  if(n==1) return 1;
  if(!DT[n]) DT[n]=2*f(n/2)+((n+1)/2)*((n+1)/2);
  return DT[n];
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

이 두 알고리즘의 동적표를 활용하고 있는 공간의 크기를 분석해보자.

n의 값이 100일 때, 두 알고리즘에서 동적표 사용량을 비교하기 위하여 동적표의 모든 데이터를 출력한 결과는 다음과 같다.

```
100
5050
0 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210 231 253 276 300
325 351 378 406 435 465 496 528 561 595 630 666 703 741 780 820 861 903 946 990
1035 1081 1128 1176 1225 1275 1326 1378 1431 1485 1540 1596 1653 1711 1770 1830
1891 1953 2016 2080 2145 2211 2278 2346 2415 2485 2556 2628 2701 2775 2850 2926
3003 3081 3160 3240 3321 3403 3486 3570 3655 3741 3828 3916 4005 4095 4186 4278
4371 4465 4560 4656 4753 4851 4950 5050
```

```
100
5050
0 0 6 0 0 21 0 0 0 0 0 78 0 0 0 0 0 0 0 0 0 0 0 0 325 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1275 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5050
```

두 동적표의 결과를 보면 위의 자료는 100개의 칸을 모두 이용한 것이고, 두 번째 알고리즘은 6개의 칸 만 이용하여 해를 구했다. 따라서 알고리즘의 효율이 두 번째 방법이 더 나음을 알 수 있다.

나. 상향식 설계

상향식 설계는 하향식 설계와는 반대로 1부터 n의 방향으로 해를 구해가는 알고리즘 설계방법을 말한다. 일반적으로 상향식 설계는 반복문으로 구현할 수 있으며, 재귀함수로는 전체탐색법에서와 같은 방법의 상향식으로 구현할 수 있다. 재귀호출을 상향식으로 작성할 때에는 $f(0)$ 또는 $f(1)$의 형태로 호출한다.

앞의 동적표를 상향식으로 채우는 방법에 대해서 알아보자. 일단 재귀함수를 이용하는 방법과 반복문을 이용하는 방법이 있다. 먼저 반복문으로 채우는 방법에 대해서 알아보자.

```
#include <cstdio>

int D[100], n;

int main()
{
  scanf("%d", &n);
  for(int i=1; i<=n; i++)
    if(i==1)
      D[i]=1;
    else
      D[i]=D[i-1]+i;
  printf("%d\n", D[n]);
  return 0;
}
```

위와 같은 알고리즘 설계법을 흔히 동적계획법(dynamic programming)이라고 한다. 이는 동적표를 이용한 설계법이라는 의미를 담고 있다.

위 알고리즘에서 다른 값의 합들이 필요 없다면 일반적으로 다음과 같이 나타낼 수 있다.

```
#include <cstdio>

int D, n;

int main()
{
  scanf("%d", &n);
  for(int i=1; i<=n; i++)
    D += i;
  printf("%d\n", D);
  return 0;
}
```

이 방법은 일반적으로 프로그래밍을 처음 배울 때, 1부터 n까지의 합을 구하는 방법이다. 이 방법은 이와 같이 동적계획법 즉, 표를 상향식으로 채우는 방법으로부터 나온 방식이다.

이와 같이 반복문을 이용하면 매우 편리하게 표를 채워서 해를 구할 수 있다. 다음으로 재귀함수를 이용하여 채우는 방법에 대해서 알아보자.

줄	코드	참고

```
#include <cstdio>

int n;

int f(int k)
{
  if(k==n+1)
    return 0;
  return k+f(k+1);
}

int main()
{
  scanf("%d", &n);
  printf("%d\n", f(1));
  return 0;
}
```

이 방법은 재귀함수로 1부터 n까지의 합을 구하는 알고리즘이다. 이 알고리즘은 중급에서 다루었던 전체탐색법과 유사한 방법이다. 다음으로 이 방법에서 동적표를 채우는 코드를 추가한 소스는 다음과 같다.

코드	참고

```
#include <cstdio>

int D[100], n;

int f(int k)
{
  if(k==n+1)
    return 0;
  D[k]=k+f(k+1);
  return D[k];
}

int main()
{
  scanf("%d", &n);
  printf("%d\n", f(1));
  return 0;
}
```

위 알고리즘에서 동적표를 채워서 활용하는 알고리즘이다. 지금 이 알고리즘은 선형 시간의 알고리즘이므로 동적표 사용여부에 따른 계산량의 변화는 없다.

하지만 비선형 알고리즘의 경우 동적표를 사용하여 계산량이 획기적으로 개선되는 예가 있으므로 익혀둘 필요가 있다.

다음 주어진 예제들을 통하여 동적표를 이용한 알고리즘을 익혀보자.

문제 1

숫자 뒤집기(L)

하나의 정수가 입력된다.

이 정수를 거꾸로 출력하는 프로그램을 작성하시오.

예를 들어 입력되는 정수가 123이라면 321을 출력하면 된다.

단, 12300이 입력될 경우 00321을 출력하는 것이 아니라 321을 출력해야 함에 주의해야 한다.

입력

첫 줄에 하나의 정수가 입력된다.
(1 <= n <= 50,000)

출력

입력된 정수를 거꾸로 출력한다.

입력 예	출력 예
123	321
12300	321

풀이

이 문제는 다양한 방법으로 해결할 수 있는 문제이지만 다음과 같은 관계식을 이용하여 상향식으로 설계할 수 있다.

먼저 함수를 다음과 같이 정의하자.

$$DT[n] \;=\; \text{“}n\text{을 거꾸로 출력한 수”}$$

이번에는 $DT[n]$에 관한 관계식을 만들어야 된다. 먼저 n이 10 미만인 경우에는 거꾸로 써도 같은 값이므로 이 값들은 관계식 없이 직접 구할 수 있다.

$$DT[n] \;=\; n \quad (n < 10)$$

위 식들을 정리하면 다음과 같은 관계식을 유도할 수 있다.

$$DT[n] \;=\; \begin{cases} n & (n < 10) \\ DT[n/10] + (n\%10) \times 10^{[\log n]} & (n \geq 10) \end{cases}$$

이 식을 이용하여 상향식으로 설계한 소스코드는 다음과 같다.

줄	코드	참고

```
#include <stdio.h>
#include <math.h>

int DT[50001];

int main()
{
  int n, i;
  scanf("%d", &n);
  for(i=1; i<=n; i++)
  {
    if(i<10)
      DT[i]=i;
    else
      DT[i]=DT[i/10]+(i%10)*(int)pow(10,(int)log10(i));
  }
  printf("%d\n",D T[n]);
  return 0;
}
```

위의 상향식 알고리즘은 매우 효율적으로 동작한다.

다음으로 하향식으로 동적표를 적용해보자. 즉 메모이제이션 기법으로 해결해보자. 이번에는 관계식(점화식)이 다음과 같다.

$$f(n) = \begin{cases} n & (n < 10) \\ f(\frac{n}{10}) + (n\%10) \times 10^{\lfloor \log n \rfloor} & (n \geq 10) \end{cases}$$

다음으로 동적표와 함수는 다음과 같이 연결시키도록 한다.

$$DT[n] = f(n)\text{의 값}$$

따라서 각 $f(k)$는 오직 한 번씩만 호출됨을 보장할 수 있다.

동적표를 적용한 하향식 알고리즘은 다음과 같다.

코드	참고

```
#include <cstdio>
#include <cmath>

int DT[50001];

int f(int n)
{
  if(n<10)
    return n;
  if(!DT[n])
    DT[n]=f(n/10)+(n%10)*powf(10.0,(int)log10((double)n));
  return DT[n];
}

int main()
{
  int n;
  scanf("%d", &n);
  printf("%d\n", f(n));
  return 0;
}
```

이와 같이 재귀함수를 이용하여 동적표를 적용할 때에는 반드시 재귀함수는 반환값을 가지는 함수이어야 한다. void 타입으로는 표에 적용하기 쉽지 않다.

그리고 이 문제의 경우에는 원래 재귀함수에서 중복호출이 발생하지 않는 구조였다. 따라서 동적표에 기록한 값을 재사용하는 일이 없기 때문에 효율이 좋아지지는 않는다.

문제를 해결할 때 재귀호출이 비선형 구조이면서 중복호출이 발생할 때에는 이와 같이 처리하면 효율이 좋아지므로 문제해결 시에 적절히 판단할 수 있도록 한다.

문제 2

combination(L)

nCk는 n개 중에서 k개를 고르는 방법의 수이다.

nCk를 구하기 위한 일반식은 다음과 같다.

$$\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1} = \frac{n!}{k!(n-k)!}$$

위 식은 n!을 이용하기 때문에 n이 커지면 overflow가 발생하여 정확한 값을 구할 수 없다.

물론 위 방법 이외에도 다양한 점화식으로도 구할 수 있다.

nCk를 정확하게 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에 n과 k가 공백으로 구분되어 입력된다.
(단, 1 <= n, k < 30)

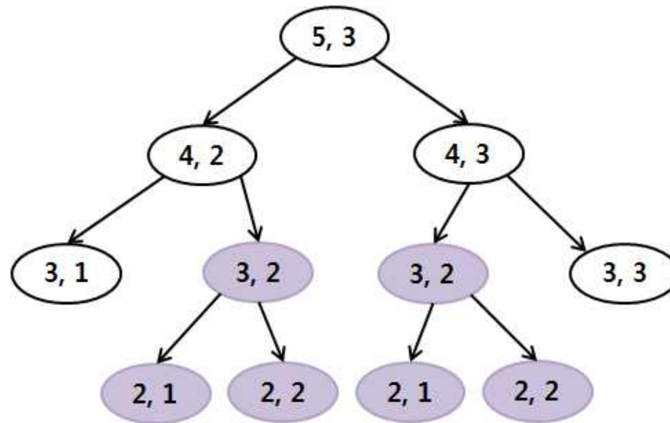
출력

구한 답을 첫 번째 줄에 출력한다.

입력 예	출력 예
5 1	5
10 5	252

풀이

먼저 이 문제를 재귀함수를 이용하여 하향식으로 설계해 보자. 5가지 중 3가지를 선택하는 경우에 대한 재귀호출의 구조는 다음과 같다.



위의 구조에서 보면 진하게 표시된 정점들은 모두 중복호출이 발생하는 정점들이다. n과 k의 값이 커질수록 중복호출의 수는 기하급수적으로 늘어난다. 따라서 동적표를 활용하면 효율이 매우 좋아질 수 있다.

따라서 다음 관계식을 이용하여 소스코드를 작성해보자.

$$f(n,\ k) = \begin{cases} 1 & (k = n) \\ n & (k = 1) \\ f(n-1,\ k-1) + f(n-1,\ k) & (1 < k < n) \end{cases}$$

위 관계식을 다음과 같이 동적표에 적용시키자. 이번의 경우는 2차원이라 다음 2가지 방법으로 적용할 수 있다.

$DT[n][k] = f(n,\ k)$의 값

$DT[nK + k] = f(n,\ k)$의 값(단, K는 최대 열의 크기)

위 관계를 이용하여 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <cstdio>

int DT[31][31];

int f(int n, int k)
{
  if(k==n) DT[n][k]=1;
  else if(k==1) DT[n][k]=n;
  else
  {
    if(!DT[n][k])
      DT[n][k]=f(n-1,k-1)+f(n-1,k);
  }
  return DT[n][k];
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  printf("%d\n", f(n,k));
  return 0;
}
```

다음으로 이 문제를 반복문을 이용하여 해결해보자. 하향식 재귀함수로 만든 관계식을 이용하면 간단하게 동적표를 설계할 수 있다.

먼저 동적표를 다음과 같이 정의하자.

$DT[n][k]$ = "n 개 중에 k 개를 고르는 경우의 수"

따라서 다음과 같은 관계를 얻을 수 있다.

$DT[n][k] =$ 마지막 물체를 고른 경우의 수 + 마지막 물체를 고르지 않은 경우의 수
마지막 물체를 고른 경우의 수 = $DT[n-1][k-1]$
마지막 물체를 고르지 않은 경우의 수 = $DT[n-1][k]$

그림을 관계식으로 나타낸 것이다. 이를 종합하면 다음과 같은 관계식을 얻는다.

$$DT[n][k] = DT[n-1][k-1] + DT[n-1][k]$$

그리고 직접 계산하는 상태는 'n개 중 n개를 고르는 경우의 수'와 'n개 중 1개를 고르는 경우의 수'로 이 값은 각각 1가지와 n가지이므로 다음과 같이 정의할 수 있다.

$$DT[1][1] = 1,\quad DT[n][n] = 1,\quad DT[n][1] = n$$

위의 관계를 정리하여 관계식을 구하면 다음과 같다.

$$DT[n][k] = \begin{cases} 1 & (k = n) \\ n & (k = 1) \\ DT[n-1][k-1] + DT[n-1][k] & (1 < k < n) \end{cases}$$

이 관계식을 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>

int DT[31][31];

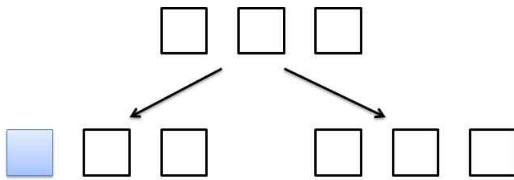
int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  for(int i=1; i<=n; i++)
  {
    for(int j=1; j<=k && j<=i; j++)
    {
      if(i==j) DT[i][j]=1;
      else if(j==1) DT[i][j]=i;
      else DT[i][j]=DT[i-1][j-1]+DT[i-1][j];
    }
  }
  printf("%d\n", DT[n][k]);
  return 0;
}
```

이 두 방법 모두 계산량은 $O(nk)$로 매우 효율적으로 동작한다. 하지만 이 두 방법으로 문제를 해결하기 위해서는 관계식(점화식)을 유도할 필요가 있다.

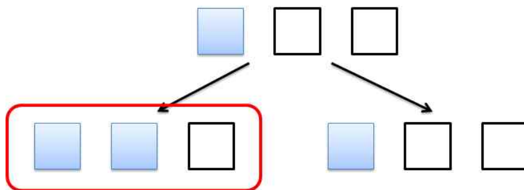
문제의 특성에 따라 관계식을 유도하기 쉽지 않은 문제들이 많다. 문제의 상황이 복잡할수록 관계를 이용한 식을 유도하기 어렵다. 즉 하향식으로 생각하기 쉽지 않다.

만약 문제의 관계식을 유도하기 쉽지 않은 경우에는 먼저 전체탐색법으로 설계를 한다. 전체탐색법은 일종의 상향식 설계방법으로 사람이 한 단계, 한 단계 직접 행동하듯이 생각하여 접근할 수 있기 때문에 비교적 설계가 단순하다.

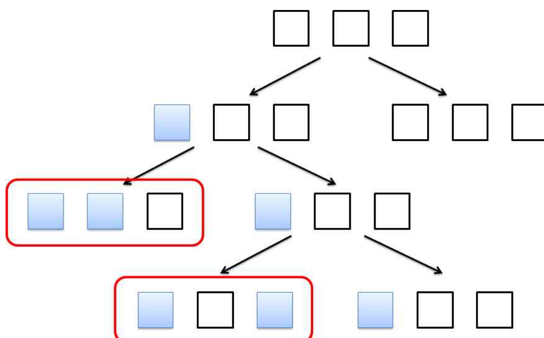
3개 중 2개를 고르는 경우에 대해서 생각해보자. 먼저 3개를 그림과 같이 한 줄로 늘어놓는다.



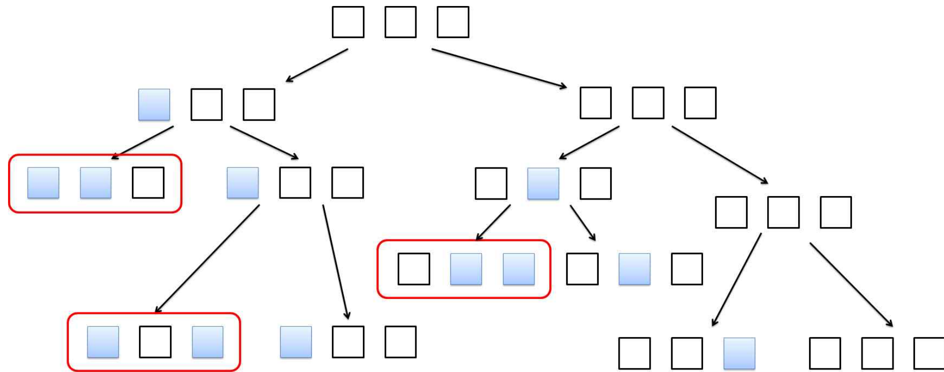
첫 번째 물건에 대해서 판단을 시작한다. 먼저 첫 번째 물건에 대해서 가능한 판단은 이 물건을 고르는 방법(왼쪽방향)과 고르지 않는 방법(오른쪽 방향)이 있다.



먼저 왼쪽 부분을 기준으로 탐색을 다시 진행한다.
이 상황에서 다시 2번째 물건을 선택하거나 선택하지 않는 방법이 있다.
여기서 왼쪽 상태에서 이미 2가지를 골랐으므로 1가지의 경우를 찾은 것이다.



지금까지 탐색한 전체 구조를 보면 다음과 같다.
현재까지 2가지의 경우를 찾았고, 이와 같이 전체탐색법으로 진행을 하면 상당히 많은 시간이 걸린다.



위와 같이 탐색하여 전체의 경우의 수를 구할 수 있다. 탐색함수 전체 물건의 수가 n개이고, 물건의 번호는 0부터 $n-1$까지일 때, $solve(n,\ k)$를 다음과 같이 설계한다.

$solve(n,\ k)$는 현재까지 k개의 물건을 고른 상태이며,
n번째 물건에 대해서 고를지 안 고를지 결정하고자 하는 상태

그리고 탐색 중단 조건은 다음과 같다.

$n\ =\ N$: 모든 물건을 탐색했는데 k개를 고르지 못한 경우(조건을 만족시키지 않음)
$k\ =\ K$: 정확히 k개의 물건을 고른 경우(조건을 만족시킴, 경우의 수 1 증가)

다음으로 $solve(n,\ k)$의 다음 상태는 다음과 같다.

$solve(n+1,\ k+1)$: 이번 물건을 선택하고 다음 물건에 대해서 탐색을 진행
$solve(n+1,\ k)$: 이번 물건을 선택하지 않고 다음 물건에 대해서 탐색을 진행

이 구조를 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>

int N, K, ans;

void solve(int n, int k)
{
  if(k==K)
  {
    ans++;
    return;
  }
  if(n==N) return;
  solve(n+1, k+1);
  solve(n+1, k);
}

int main()
{
  scanf("%d %d", &N, &K);
  solve(0,0);
  printf("%d\n", ans);
  return 0;
}
```

위 소스코드는 중급 편에서 다루었던 전체탐색법을 구현한 것이다. 이 방법을 백트래킹이라고 한다. 백트래킹은 대부분의 문제에 대해서 해를 구할 수는 있지만 속도가 너무 느리기 때문에 그대로 활용하기에는 무리가 있는 방법이다.

이 백트래킹 기법에 동적표를 적용해보자. 흔히 이러한 방법을 다이나미컬 백트래킹(dynamical backtracking)이라고 한다.

동적표를 적용하기 위해서는 먼저 $solve$ 함수를 void가 아닌 값을 반환하는 함수 f로 수정해야 한다. 수정한 결과는 다음과 같다.

```
#include <cstdio>

int N, K;

int f(int n, int k)
{
  if(k==K) return 1;
  if(n==N) return 0;
  return f(n+1,k+1)+f(n+1,k);
}

int main()
{
  scanf("%d %d", &N, &K);
  printf("%d\n", f(0,0));
  return 0;
}
```

이 방법에서 함수를 살펴보면 $f(0,\ 0)$으로부터 출발하므로 상향식 기법이라고 할 수 있다. 이제 f 함수의 값이 하나의 정수를 반환하므로 동적표를 적용할 수 있다. 동적표는 다음과 같이 정의한다.

$$DT[n][k] \ = \ f(n,\ k)\text{의 값}$$
$$DT[nK \ + \ k] \ = \ f(n,\ k)\text{의 값(단, }K\text{는 최대 열의 크기)}$$

이 표를 적용한 소스코드는 다음과 같다.

```
#include <cstdio>

int N, K, DT[31][31];

int f(int n, int k)
{
  if(k==K) return 1;
  if(n==N) return 0;
  if(!DT[n][k])
    DT[n][k]=f(n+1,k+1)+f(n+1,k);
  return DT[n][k];
}

int main()
{
  scanf("%d %d", &N, &K);
  printf("%d\n", f(0,0));
  return 0;
}
```

이 방법은 앞에서 다루었던 동적계획법과 같은 속도로 동작한다. 따라서 매우 효율적인 방법이라고 할 수 있다. 특히 점화관계를 유도하기 어려운 문제들을 백트래킹으로 처리할 수 있기 때문에 실제 대회에서 자주 응용할 수 있는 기법이므로 반드시 익혀둘 수 있도록 한다.

다이나미컬 백트래킹의 가장 큰 장점은 설계는 백트래킹으로 접근하고 효율은 동적계획법의 효율이 난다는 점이다. 단 동적계획법으로 해결했을 때와 다이나이컬 백트래킹으로 해결했을 때의 동적표에 채워진 값의 방향이 반대라는 점이 흥미롭다.

두 가지 방법으로 10개의 중 5개를 택한 문제를 해결한 후의 동적표의 값은 다음과 같다.

252

1	0	0	0	0
2	1	0	0	0
3	3	1	0	0
4	6	4	1	0
5	10	10	5	1
6	15	20	15	6
0	21	35	35	21
0	0	56	70	56
0	0	0	126	126
0	0	0	0	252

[동적계획법으로 채운 동적표]

252

252	0	0	0	0
126	126	0	0	0
56	70	56	0	0
21	35	35	21	0
6	15	20	15	6
1	5	10	10	5
0	1	4	6	4
0	0	1	3	3
0	0	0	1	2
0	0	0	0	1

[다이나미컬 백트래킹으로 채운 동적표]

이와 같이 두 표의 방향이 서로 반대이다. 어떤 문제들의 경우 경로를 추적하는 경우의 문제가 있는데 만약 동적계획법으로 표를 채우면 거꾸로 탐색해야 하지만 다이나미컬 백트래킹으로 채우면 경로의 추적 또한 상향식으로 가능하므로 편리한 점이 있다.

문제 3

숙직 선생님

학교에 몰래 누군가 침투를 했다. 그래서 학교 숙직 선생님이 누군가를 잡기 위해 찾으러 다닌다. 이때 숙직 선생님의 현재 위치와 침투한 누군가의 위치가 주어질 때, 숙직 선생님은 모두 다른 특별한 능력을 3가지 받게 된다.

이 능력은 1초에 이동할 수 있는 거리를 이야기 한다. 1초에 하나의 능력만 사용이 가능하다. 능력을 이용하여 침투한 누군가를 가장 빨리 찾을 수 있는 최소 시간을 출력하라.

경우에 따라서는 찾지 못할 수도 있다. 찾지 못할 경우 -1을 출력한다. 단, 겁이 많은 누군가는 움직이지 않고 숨어만 있다. 이동은 위치가 커지는 방향으로 이동한다.

입력

첫 번째 줄에 숙직 선생님의 현재위치 a 누군가의 위치 b(1<=a<=b<=1000 인 정수)

두 번째 줄에 3개의 숙직 선생님이 사용할 수 있는 능력이 입력된다(1<= 능력 <=100 인 정수)).

출력

누군가를 찾는 데 걸리는 시간(초)를 출력한다.

단, 찾지 못할 경우는 –1을 출력한다.

입력 예	출력 예
1 15 2 5 7	2

풀이

먼저 전체탐색법으로 해결하는 방법부터 알아보자.

현재 숙직 선생님의 위치부터 시작해서 갈 수 있는 능력으로 전부 다 방문해 보는 것이다. 현재 위치에서 갈 수 있는 세 가지 위치로 가면서 시간을 누적해서 더해 가는 것이다.

```
#include <stdio.h>
int a, b, able[3], ans=2000;

void back(int next, int cnt)
{
  if(next>b) return;
  if(b==next)
  {
    if(cnt<ans) ans=cnt;
  }
  for(int i=0; i<3; i++)
    back(next+able[i], cnt+1);
}

int main()
{
  scanf("%d %d", &a, &b);
  for(int i=0; i<3; i++)
    scanf("%d", &able[i]);
  back(a, 0);
  if(ans!=2000) printf("%d", ans);
  else printf("-1");
  return 0;
}
```

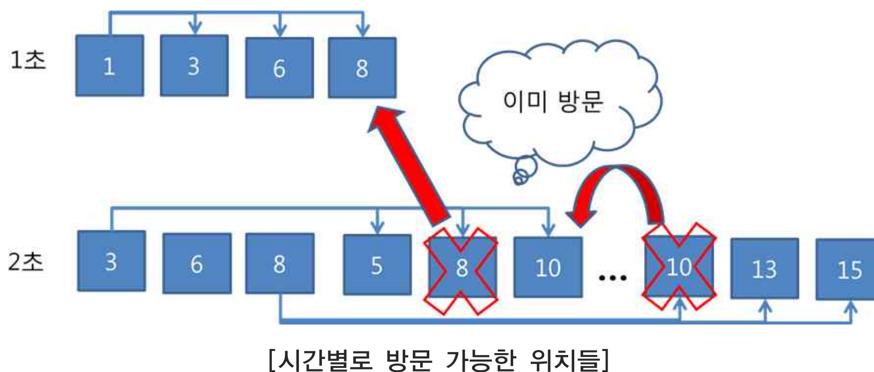
이 소스코드는 모든 가능한 경우를 전부 탐색하기 때문에 계산량은 $O(3^n)$이다. 따라서 주어진 조건에서 탐색하는 경우 시간이 너무 많이 걸린다. 따라서 탐색영역을 배제할 필요가 있다.

목표지점을 찾을 경우 소요시간을 기록하고 소요시간이 초과 되는 경우 종료조건을 추가하면 시간을 조금 줄일 수 있다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`int a, b, able[3], ans=2000;`	
3	`void back(int next, int cnt)`	
4	`{`	
5	`    if(ans<cnt) return;`	
6	`    if(next>b) return;`	

그러나 이 방법으로는 주어진 조건이 커질 경우는 별다른 효과가 없다.

다른 아이디어가 필요하다. 아래 그림과 같이 현재 숙직 선생님의 위치에서 갈 수 있는 능력은 세 가지이므로 현재 위치에서 세 가지 능력으로 도착하는 위치는 1초에 갈 수 있다. 한 위치에서 세 가지 위치로 이동이 되므로 현재까지 소요시간 +1씩 해서 방문 표시를 하게 되면 원하는 위치를 계산할 수 있다. 이때 아래 그림과 같이 해당 지점을 방문했는지를 체크하여 다음 위치로 이동하면 된다. 이때 자료구조인 큐를 이용하면 쉽게 해결할 수 있다.



[시간별로 방문 가능한 위치들]

```
#include <stdio.h>
#include <queue>
using namespace std;
queue<int> q;
int d[1200];

int main()
{
  int a, b, able[3];
  scanf("%d%d", &a, &b);
  for(int i=0; i<3; i++)
    scanf("%d", &able[i]);
  q.push(a);
  while(!q.empty())
  {
    int value=q.front();
    if(value==b)
    {
      printf("%d", d[value]);
      return 0;
    }
    q.pop();
    for(int i=0; i<3; i++)
    {
      int temp=value+able[i];
      if(temp<=b && !d[temp])
      {
        q.push(temp);
        d[temp]=d[value]+1;
      }
    }
  }
  printf("-1");
  return 0;
}
```

위의 알고리즘을 적용했을 때, a와 b 사이의 거리를 n이라고 하면 $O(n$*능력수$)$ 의 시간이 소요된다.

위의 소스코드를 응용하면 동적계획법으로 코딩이 가능하다. 특정 위치까지 소요시간은 (특정 위치 – 세 가지 능력)의 소요시간 중 최소 +1이 된다. 이를 바탕으로 다음과 같은 점화식을 세울 수 있다.

$D(n)$: n까지 오는 최소 소요시간
$D(a)$ = 0 (a는 시작위치)
$D(n)$ = Min($D(n-able[i])$)+1 (단, $D(n-able[i])$ > a)

아래의 그림과 같이 8번 위치까지 소요시간은 1번 위치에서 3번째 능력으로 오는 경우, 3번 위치에서 2번째 능력으로 오는 경우, 6번 위치에서 1번 능력으로 오는 경우에서 올 수 있다. 이 중에서 가장 작은 소요시간은 1번 위치에서 3번째 능력으로 오는 경우이다. 따라서 8번 위치까지 최소 소요시간은 1번 위치의 소요시간 +1이 되어 2가 된다. 이러한 방법으로 나머지 표를 완성하면 우리가 원하는 b번 위치에서 소요시간을 동적표를 이용하여 계산할 수 있다.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	∞	1	∞	2	1	3																		

[8번 위치에서 동적표 갱신 방법]

풀이는 다음과 같다. 처음에 동적표를 구할 수 없는 큰 숫자로 설정한다.

```c
#include <stdio.h>
#define MIN(a,b) ((a)>(b)?(b):(a))
int a, b, able[3], d[1001];
int main()
{
  scanf("%d%d", &a, &b);
  for(int i=0; i<3; i++)
    scanf("%d", &able[i]);

  if(a==b) printf("0");
  else
  {
    for(int i=a; i<=b; i++)
      d[i]=123456789;
    d[a]=0;

    for(int i=a+1; i<=b; i++)
    {
      int temp=123456789;
      for(int k=0; k<3; k++)
      {
        if(i-able[k]>=a)
          temp=MIN(temp, d[i-able[k]]+1);
      }
      d[i]=temp;
    }

     if(d[b]!=123456789) printf("%d", d[b]);
     else printf("-1");
  }
  return 0;
}
```

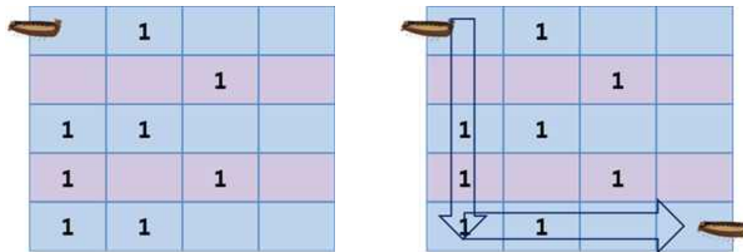
위 알고리즘의 계산량은 앞의 알고리즘과 동일하게 $O(n*$능력수$)$이지만 배열의 값만 갱신하기 때문에 조금 더 빠르다.

실제 본 문제는 위치가 1~1000이상의 더 큰 범위의 문제도 빠르게 해결할 수 있다.

문제 4

광석수집

GT혹성에 광석을 수집하는 SCV는 항상 오른쪽이나 아래쪽으로만 진행된다. 이때 SCV가 수집하는 최대 광석의 개수를 출력하자. 최대 n*m으로 이루어진 광석을 캐는 지역이 입력되고 항상 시작은 1,1에서 시작하여 n, m에서 종료된다.



위의 그림에서 SCV는 1,1에서 출발하여 그림처럼 광석을 수집하면 최대 4개가 가능하다.

입력

입력으로 광석의 최대 세로크기(1<=n<=200)와 가로크기(1<=m<=200)가 입력된다.
다음 n줄에 걸쳐 m열만큼 공백으로 구분하여 광석이 표시된다.
이때 1이 광석을 표현한다.

출력

SCV가 수집하는 최대 광석의 개수를 출력하라.

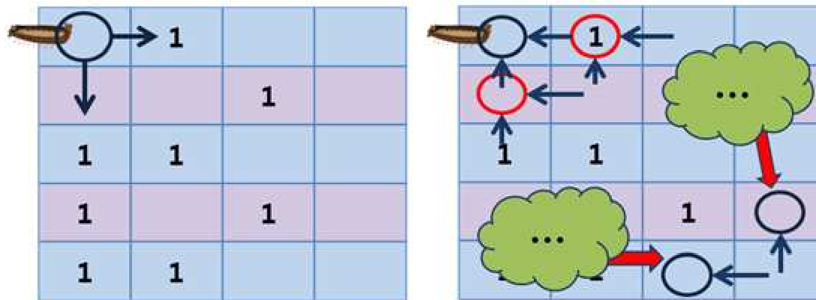
입력 예	출력 예
5 4 0 1 0 0 0 0 1 0 1 1 0 0 1 0 1 0 1 1 0 0	4

풀이

현재 위치에서 갈 수 있는 방법은 두 가지가 가능하다. 오른쪽이거나 아랫방향이다. 결국 오른쪽 방향으로 가는 경우와 아랫방향으로 가는 두 가지 방법 중에서 가장 큰 광석을 모은 방법을 택하면 된다.

따라서 이 아이디어를 살펴보면 현재 위치에서 오른쪽이나 아래쪽으로 이동하면서 광석이 있으면 누적해 간다. 이를 응용하면 현재 위치를 구하기 위해서는 아래쪽과 오른쪽 중에서 광석수집이 많이 된 것을 찾아 현재 위치의 광석의 개수를 더하며 된다.

(1,1)에서 보면 (2,1)과 (1,2)로 가게 된다. 이를 반대로 생각하면 (1,1)로 오는데 까지 광석수집의 최댓값은 (2,1)과 (1,2) 중에서 큰 값을 계산하면 된다.



이를 바탕으로 하향식 설계 방법을 작성해 보면

종료조건 : row == n and col ==m (목적지 도착 시)
w>n or col>m (맵을 벗어났을 때)

관계식 : f(n,m) = 광석맵[row][col] + MAX(f(n+1, m), f(n, m+1))

풀이는 다음과 같다.

```
#include <stdio.h>
#define MAX(a,b) ((a)>=(b)?(a):(b))

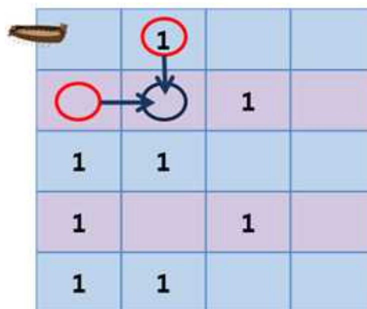
int n, m, mine[220][220], table[220][220];

int back(int row, int col)
{
  if(table[row][col]) return table[row][col];
  if(row==n && col==m) return mine[row][col];
  if(row>n || col>m) return 0;
  return table[row][col]=mine[row][col]+MAX(back(row+1,col),back(row,col+1));
}

int main()
{
  scanf("%d %d", &n, &m);
  for(int i=1; i<=n; i++)
    for(int k=1; k<=m; k++)
      scanf("%d", &mine[i][k]);
  printf("%d", back(1,1));
  return 0;
}
```

이 소스코드는 모든 표를 한번만 검색하기 때문에 계산량은 $O(nm)$ 이다.

여기서 생각의 전환을 해서 다르게 바라보면 현재 위치로 오는 경우를 생각해서 문제를 접근하면 상향식 설계가 가능하다. 동적표를 빠르게 갱신 가능하게 된다.



위의 그림과 같이 파란색 지점까지의 최대 광석수집은 위쪽에서 구한 광석수집 결과와 왼쪽에서 수집한 광석수집 결과 중에 큰 것 + 현재 지점의 광석유무를 계산하면 된다.

점화식을 정의해 보면 아래와 같다.

$$T[i][j] = \begin{cases} 0 & (i=0,\ j=0) \\ mine[i][j] + \max(T[i-1][j],\ \ T[i]][j-1] \end{cases}$$

줄 | 코드 | 참고

```
#include<stdio.h>
#define MAX(a,b) ((a)>=(b)?(a):(b))

int n, m, mine[220][220], table[220][220];

int main()
{
  scanf("%d %d", &n, &m);

  for(int i=1; i<=n; i++)
    for(int k=1; k<=m; k++)
      scanf("%d", &mine[i][k]);

  for(int i=1; i<=n; i++)
  {
    for(int k=1; k<=m; k++)
    {
      int a=table[i-1][k];
      int b=table[i][k-1];
      table[i][k]=mine[i][k]+MAX(a,b);
    }
  }
  printf("%d", table[n][m]);
  return 0;
}
```

이 소스코드는 모든 표의 크기가 n, m 이므로 이를 반복문으로 한 번만 검색하기 때문에 계산량은 $O(nm)$ 이다.

문제 5

0/1 배낭 문제(L)

어떤 배낭에 W무게만큼 물건을 담을 수 있다.

물건들은 (무게 wi, 가격 vi) 정보를 가지고 있는데, 물건들을 조합해서 담아 가격의 총합이 최대가 되게 하려고 한다.

물건들은 한 종류씩 밖에 없으며, 절대 배낭의 무게를 초과해서는 안 된다.

입력

첫째 줄에 물건의 개수 n(1<= n <= 100)과 배낭의 무게 w(1 <= w <= 10000)가 입력된다.

둘째 줄부터 n+1째줄 까지 물건들의 정보가 wi, vi가 한 줄에 하나씩 입력된다.

(1 <= wi, vi <= 100)

출력

배낭의 무게 W를 초과하지 않으면서 물건의 가격의 총합의 최댓값을 출력한다.

입력 예	출력 예
4 5 2 3 1 2 3 3 2 2	7

풀이

배낭 문제는 정보과학에 있어 유명한 문제 중 하나이다. 이 문제를 전체탐색에서부터 접근해 보자. 문제의 상황을 표로 작성해보자.

물건 번호(i)	무게(wi)	가치(vi)
1	2	3
2	1	2
3	3	3
4	2	2

배낭의 무게가 5이므로, 5를 넘지 않으면서 채울 수 있는 방법 중 최고 가치를 가지는 경우를 전체 탐색하면 된다. 이 경우 1, 2, 4번의 물건을 담는 경우 W=5가 되고, 가치는 3 + 2 + 2 = 7로 최대가 된다. 우선 백트래킹에 의한 전체 탐색을 해보자.

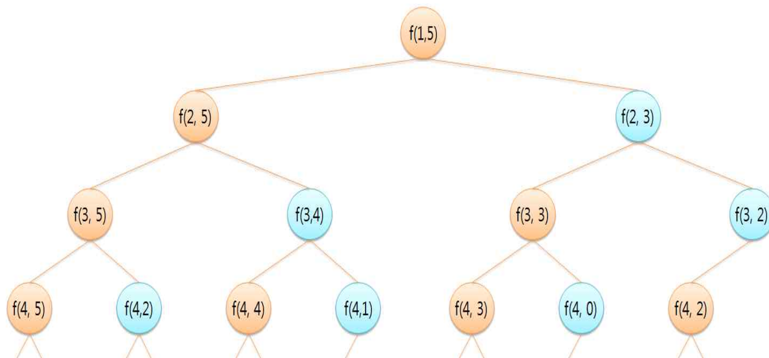
함수 f의 의미는 다음과 같다.

f(i, r) = i~n번째 물건까지 고려했을 때, 남은 무게가 r인 가치의 최대 합

이 식을 수학적으로 정리하면 다음과 같다.

$$f(i,r) = \begin{cases} \max(f(i+1,r) \text{ or } f(i+1,r-w[i])+v[i]) & (w[i] \le r) \\ f(i+1,r) & (w[i] > r) \end{cases}$$

이 함수를 이용해 실제로 호출되는 과정은 다음과 같다.



이 트리에서 물건의 개수 n이 1증가할 때마다 트리의 깊이가 1씩 증가하므로, n이 커지면 계산량이 기하급수적으로 많아진다. 이 알고리즘은 i번째 물건을 넣느냐, 넣지 않느냐로 나뉘기 때문에 전체 계산량은 $O(2^n)$ 이다.

위 알고리즘은 함수가 재귀호출되는 과정에서 여전히 중복된 연산을 수행할 수 있으므로 메모이제이션 기법을 활용하여 중복된 호출이 일어날 경우 그 결과를 재활용하여 단 번에 구하는 방법으로 계산량을 줄일 수 있다.

줄	코드	참고

```
1   #include <stdio.h>
2
3   int DT[102][10002];
4   int W, n, i, j, w[102], v[102];
5
6   int max(int a, int b) {return a>b?a:b;}
7
8   int f(int i, int r)
9   {
10    if(DT[i][r]!=-1) return DT[i][r];
11    if(i==n+1) return DT[i][r] = 0;
12    else if (r<w[i]) return DT[i][r] = f(i+1,r);
13    else return DT[i][r]=max(f(i+1,r), f(i+1,r-w[i])+v[i]);
14  }
15
16  int main()
17  {
18    scanf("%d %d", &n, &W);
19    for(i=1; i<=n; i++)
20      scanf("%d%d", &w[i], &v[i]);
21
22    for(i=0; i<=100; i++)
23      for(j=0; j<=10000; j++)
24        DT[i][j]=-1;
25
26    printf("%d", f(1,W));
27  }
```

참고: 8: f(물건 번호 i, 남은 무게 r)

3행은 계산 결과를 저장하는 동적표를 선언한 부분이다. DT[i][r]의 의미는 함수 f의 의미와 같다.

DT[i][r] = i~n번째 물건까지 고려했을 때, 남은 무게가 r인 가치의 최대 합

8~14행에서 계산의 결과를 모두 동적표 DT배열에 저장한 다음 리턴하는 것을 볼 수 있다.

동적표 DT의 사용으로 계산량은 얼마나 줄었는지 확인해보자. 재귀함수의 메모이제이션 기법으로 같은 인자로 두 번째 호출을 하는 경우 바로 결과를 얻을 수 있으므로, 속도를 확실히 줄일 수 있다. 이 알고리즘의 계산량은 $O(nW)$ 이다. 이전 알고리즘은 $O(2^n)$ 이므로, 데이터 개수에 따른 계산량을 비교해보자. 단, 여기서 W의 최대 범위는 10,000이다.

n의 크기	$O(2^n)$	$O(nW)$
10	210 = 1024	10*10000 = 100,000
50	250 = 1,125,899,906,842,620	50*10000 = 500,000
100	2100 = ???	100*10000 = 1,000,000

n이 커지면 $O(nW)$ 알고리즘의 성능이 우수함을 확실히 알 수 있다.

이 알고리즘을 상향식으로 설계해보자.

```
#include <stdio.h>

int DT[102][10002];
int w[102], v[102], i, j, n, W;

int max(int a, int b) {return a>b ? a:b;}

main()
{
  scanf("%d %d", &n, &W);
  for(i=1; i<=n; i++)
    scanf("%d %d", &w[i], &v[i]);

  for(i=n; i>=1; i--)
  {
    for(j=0; j<=W; j++)
      if(j<w[i]) DT[i][j]=DT[i+1][j];
      else DT[i][j]=max(DT[i+1][j],DT[i+1][j-w[i]]+v[i]);
  }
  printf("%d", DT[1][W]);
}
```

재귀함수를 이용하지 않더라도 동적표를 채우는 것은 중첩 반복문을 이용한 n과 W의 관계로 쉽게 채울 수 있다.

문제 6

앱(L)

우리는 스마트폰을 사용하면서 여러 가지 앱 (App)을 실행하게 된다. 대개의 경우 화면에 보이는 ‘실행중’인 앱은 하나뿐이지만 보이지 않는 상태로 많은 앱이 '활성화' 되어 있다.

앱들이 활성화되어 있다는 것은 화면에 보이지 않더라도 메인메모리에 직전의 상태가 기록되어 있는 것을 말한다. 현재 실행중이 아니더라도 이렇게 메모리에 남겨두는 이유는 사용자가 이전에 실행하던 앱을 다시 불러올 때에 직전의 상태를 메인 메모리로부터 읽어 들여 실행 준비를 빠르게 마치기 위해서이다.

하지만 스마트폰의 메모리는 제한적이기 때문에 한 번이라도 실행했던 모든 앱을 활성화된 채로 메인메모리에 남겨두다 보면 메모리 부족 상태가 오기 쉽다. 새로운 앱을 실행시키기 위해 필요한 메모리가 부족해지면 스마트폰의 운영체제는 활성화 되어 있는 앱들 중 몇 개를 선택하여 메모리로부터 삭제하는 수밖에 없다.

이러한 과정을 앱의 ‘비활성화’라고 한다. 메모리 부족 상황에서 활성화되어 있는 앱들을 무작위로 필요한 메모리만큼 비활성화하는 것은 좋은 방법이 아니다.

비활성화된 앱들을 재실행할 경우 그만큼 시간이 더 필요하기 때문이다. 여러분은 이러한 앱의 비활성화 문제를 스마트하게 해결하기 위한 프로그램을 작성해야 한다.

현재 n개의 앱 A1, ... ,An이 활성화되어 있다고 가정하자. 이들 앱 Ai는 각각 mi 바이트만큼의 메모리를 사용하고 있다.

또한, 앱 Ai를 비활성화한 후에 다시 실행하고자 할 경우, 추가적으로 들어가는 비용(시간 등)을 수치화한 것을 ci라고 하자. 이러한 상황에서 사용자가 새로운 앱B를 실행하고자 하여, 추가로 M 바이트의 메모리가 필요하다고 하자.

앱(L) (계속)

즉, 현재 활성화되어 있는 앱 A1, ..., An 중에서 몇 개를 비활성화하여 M 바이트 이상의 메모리를 추가로 확보해야 하는 것이다. 여러분은 그 중에서 비활성화했을 경우의 비용 ci의 합을 최소화하여 필요한 메모리 M 바이트를 확보하는 방법을 찾아야 한다.

입력

첫 줄에는 정수 n과 M이 공백문자로 구분되어 주어지며,

둘째 줄과 셋째 줄에는 각각 n개의 정수가 공백문자로 구분되어 주어진다.

둘째 줄의 n개의 정수는 현재 활성화되어 있는 앱 A1,..., An이 사용 중인 메모리의 바이트 수인 m1, ..., mn을 의미하며,

셋째 줄의 n 개의 정수는 각 앱을 비활성화했을 경우의 비용 c1,..., cn을 의미한다.

[입력의 정의역]

1 ≤ N ≤ 100 ; 1 ≤ M ≤ 10,000,000 ; 1 ≤ m1, ..., mn ≤ 10,000,000

0 ≤ c1,.., cn≤ 100

출력

필요한 메모리 M 바이트를 확보하기 위한 앱 비활성화의 최소의 비용을 계산하여 한 줄에 출력해야 한다.

입력 예	출력 예
5 60 30 10 20 35 40 3 0 3 5 4	6

출처: 한국정보올림피아드(2013 지역본선 고등부)

풀이

이 문제는 새로운 앱을 실행하기 위해 활성화되어 있는 앱들 중 몇 개를 비활성화해서 메모리 M이상을 확보하는 데 드는 비용을 최소화하는 문제이다.

이 상황을 잘 생각해보면 배낭 문제와 유사해 보인다.

	배낭 문제		앱
N	배낭에 담을 수 있는 물건 개수	n	비활성화 할 수 있는 앱의 개수
W	배낭의 무게	M	확보해야할 메모리량
Wi	각 물건의 무게	mi	각 앱의 메모리 사용량
Vi	물건의 가치	ci	앱의 비활성화에 드는 비용

배낭 문제에서는 배낭의 무게 W를 넘지 않으면서 물건 가치를 최대로 높이는 경우를 찾는 것이고, 앱 문제에서는 메모리를 M이상을 확보하면서 비활성화에 드는 최소 비용을 찾는 것이다.

문제에서 제시한 상황은 다음 표와 같다.

앱의 번호(i)	사용 중인 메모리(mi)	비활성화 비용(ci)
1	30	3
2	10	0
3	20	3
4	35	5
5	40	4

요구하는 메모리가 60이므로 비활성화 비용을 최소화하면서 60 이상의 메모리를 확보하기 위해서는 1, 2, 3번의 앱을 비활성화 시켜야 한다.(30+10+20=60, 3+0+3=6)

배낭 문제와 유사하기 때문에 배낭 문제에서 사용한 알고리즘을 이 문제에 맞게 변형시켜보자. 배낭문제에서는 f(1, 0)을 호출해서 답을 구했지만, 이번에는 다른 방법으로 설계해도 똑같은 결과가 나온다는 것을 보여주기 위해 반대로 f(n, M)으로 설계하였다.

함수 f의 의미는 다음과 같다.

f(i, r) = 1~i번째 앱까지 고려했을 때, 메모리 r이상 확보하기 위한 최소 비용

설계 방법의 변경과 최솟값을 구하는 부분에서 약간의 소스가 변경되었다.

줄	코드	참고

```
1   #include <stdio.h>
2   #define MAXV 999999
3
4   int M, n, i, m[101], c[101];
5
6   int min(int a, int b) {return a<b ? a:b;}
7
8   int f(int i, int r)
9   {
10    if(i==0)
11    {
12      if(r<=0) return 0;
13      else return MAXV;
14    }
15    else if(r<0) return f(i-1,r);
16    else  return min(f(i-1,r), f(i-1,r-m[i])+c[i]);
17  }
18
19  int main()
20  {
21    scanf("%d %d", &n, &M);
22    for(i=1; i<=n; i++)    scanf("%d",&m[i]);
23    for(i=1; i<=n; i++)    scanf("%d",&c[i]);
24    printf("%d",f(n,M));
25  }
```

참고: 8: f(앱 번호 i, 남은 메모리 r)

이 알고리즘은 앱의 개수 n에 의해 계산량이 결정되는 $O(2^n)$ 알고리즘이므로 n제한이 100인 이 문제를 제한 시간 안에 해결할 수 없다. 따라서 메모이제이션 기법이나 동적계획법을 사용해야 한다. 위 풀이에 메모이제이션 기법을 적용해보자.

int DT[101][10000001];

동적표를 저장하는 2차원 배열 DT를 선언하려고 보니 배열의 크기에 실행이 될지 의문이 생긴다. 입력의 정의역의 최대 크기에 따라 실제 크기를 계산해보면, 101 × 10,000,001 × 4 = 4,040,000,404바이트이다. 현재 사용되는 컴퓨터들 중 32비트 컴퓨터에서는 실행이 불가능한 프로그램이다.

그럼 이 문제를 어떻게 해결해야 할까? 지금까지 i번째 앱에 메모리 사용량이 r이 되는 최소 비용을 탐색하였다. 이것을 조금 바꿔 생각해보자. 이 문제를 활성화되어 있는 앱들 중 몇 개를 선택하고, 그들의 비용을 기준으로 탐색한다면, 가능한 최대의 메모리 합이 M이상이 되는 최소 c값을 구하는 문제로 바꿀 수 있다.

DT[i][j] = 1~i번째 앱까지 고려했을 때, 비용의 합이 j인 최대 메모리의 합

$$DT[i][j] = \begin{cases} \max(DT[i-1][j],\ DT[i-1][j-c[i]]+m[i]) & (j \geq c[i]) \\ DT[i-1][j] & (j < c[i]) \end{cases}$$

모든 동적표를 계산한 후,
DT[n][0] ~ DT[n][n×100]을 탐색하여, 메모리의 합이 M이상인 최소 비용 j를 찾는다.

위 점화식으로 샘플 예제를 동적표로 작성해보자.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	30	30	30	30	30	30	30	30	30	30	30	30	30
2	10	10	10	40	40	40	40	40	40	40	40	40	40	40	40	40
3	10	10	10	40	40	40	60	60	60	60	60	60	60	60	60	60
4	10	10	10	40	40	45	60	60	75	75	75	95	95	95	95	95
5	10	10	10	40	50	50	60	80	80	85	100	100	115	115	115	135

DT[2][3]의 의미는 1~2번째 앱을 고려했을 때, 비용의 합이 3인 경우, 최대 메모리 합이 40이라는 것을 의미한다. 이것은 DT[1][3]과 DT[1][3-0] + 10의 값 중 큰 값으로 결정한다.

이렇게 동적표를 구한 다음 마지막에 i의 값이 5이면서, j의 값이 60을 넘는 수 중 가장 작은 j값을 찾으면 최소 비용이 된다. 이 점화식을 적용한 상향식 알고리즘 소스이다.

```c
#include <stdio.h>

int n, M, m[101], c[101];
int DT[101][10001];
int max(int a, int b) {return a>b ? a:b;}
int main()
{
  int i, j, res=100000;
  scanf("%d %d", &n, &M);
  for(i=1; i<=n; i++) scanf("%d", &m[i]);
  for(i=1; i<=n; i++) scanf("%d", &c[i]);

  for(i=1; i<=n; i++)
    for(j=0; j<=100*n; j++)
      if(j>=c[i])
         DT[i][j]=max(DT[i-1][j], DT[i-1][j-c[i]]+m[i]);
      else
         DT[i][j]=DT[i-1][j];

  for(j=0; j<=100*n; j++)
    if(DT[n][j]>=M && res>j)
      res = j;

  printf("%d", res);
}
```

생각을 조금만 바꾸면 불가능해 보이는 문제도 해결할 수 있다. 문제에 따라 배열 크기 제한에 걸리는 경우 알고리즘을 변경해야 한다.

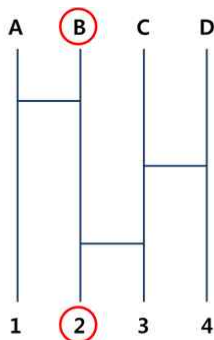
이 알고리즘의 최대 계산량은 n개의 앱과 그 비용의 곱으로 표현되므로, 100 * 10,000 = 1,000,000 이다. 이는 제한시간에 충분히 해결할 수 있다. 이 알고리즘의 전체적인 계산량은 $O(n^2)$ 으로 나타낼 수 있다.

문제 7

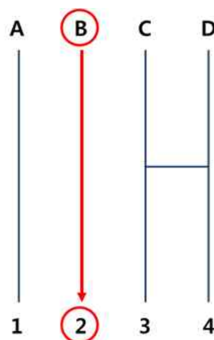
잭과 콩나물

최근의 극장가에 '잭 더 자이언트 킬러'가 상영되었다. 이 영화 속 '클로이스트'의 시골 농장에서 삼촌과 함께 살고 있는 잭은 시장에 말을 팔러 갔다가 돈 대신 콩 몇 알을 얻게 된다. 그날 밤 잭의 집으로 세찬 비바람을 피해 낯선 손님 이자벨이 찾아온다. 그리고 우연히 잭이 낮에 얻어온 콩이 물에 젖어 하늘로 뻗어 오르면서 이야기가 시작된다.

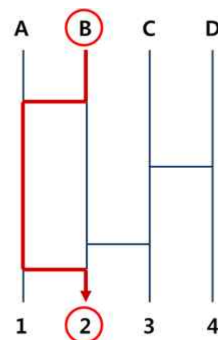
이 영화를 본 E교수는 학생들에게 콩나물을 연결하는 문제를 제시하였다. 세로 형태의 콩나물이 n개가 있고, 세로 콩나물 사이에 가로 형태의 콩나물 m개가 설치되어 있다. 세로 형태의 콩나물은 제일 왼쪽부터 번호가 1, 2, ..., n이고, 가로 형태의 콩나물은 제일 위에서부터 번호가 1, 2, ..., m이다. 같은 높이에 위치한 가로 콩나물은 없다.



〈그림1〉



〈그림2〉
제거한 경우



〈그림3〉
추가한 경우

<그림1>과 같은 경우 n=4, m=3인 콩나물들이다. A는 3으로, B는 1로, C는 4로, D 는 2로 가게 된다. 이는 사다리게임과 동일하다. 우리는 이 콩나물들을 조작해서 우리가 원하는 모양으로 만들고자 한다.

잭과 콩나물 (계속)

콩나물을 조작할 때에는 가로 형태의 콩나물만 제거하거나 추가할 수 있다. 콩나물 하나를 제거할 때 X만큼의 비용이 들고 추가할 때 Y만큼의 비용이 필요하다. 예를 들어 <그림1>의 경우 B에서 2로 가고자 한다면 <그림2>처럼 A-B 사이에 연결된 콩나물과 B-C 사이에 연결된 콩나물을 제거하거나 <그림3>처럼 B-C 사이에 연결된 콩나물 밑에 A-B 사이에 연결하는 콩나물을 추가하면 된다.

만약, 제거하는 비용이 1이고, 추가하는 비용이 5라면 <그림2>처럼 2개를 제거한 경우에는 2의 비용이 들고 <그림3>처럼 1개를 추가할 경우에는 5의 비용이 든다. 그러므로 최소 비용은 2가 된다. 위와 같이 출발점 a 위치에서 아래의 도착점 b 위치로 갈 수 있도록 조작할 때 필요한 최소 비용을 구하는 프로그램을 작성하시오.

입력

1. 첫째 줄에 n, m이 주어진다. (단, 1≦n≦100, 1≦m≦500)
2. 둘째 줄에는 a, b, X, Y가 주어진다(단, 0≦X≦1000, 0≦Y≦1000).
3. 셋째 줄부터 m +2줄까지는 위에서부터 가로 콩나물에 대한 정보를 나타내는 정수 p가 주어진다. p의 의미는 p번과 p +1번의 세로 콩나물을 연결하는 가로 콩나물을 의미한다.

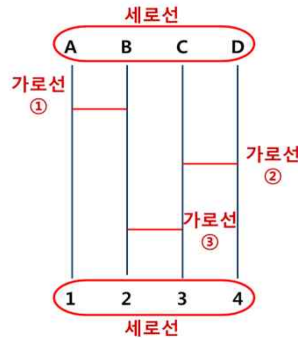
출력

첫 줄에 최소 비용을 출력한다.

입력 예	출력 예
4 3 2 2 1 5 1 3 2	2

풀이

문제를 정확하게 이해하고 점화식을 세워 해결해야 되는 동적계획법 문제이다. 입력에 대한 정의역의 크기만 보아도 단순한 전체 탐색으로는 해결할 수 없는 문제이다. 입출력 예시에 나온 예제를 기준으로 설명하되, 명확한 설명과 단순화를 위해 가로 콩나물, 세로 콩나물이란 말 대신 가로선과 세로선으로 나눠 설명할 수 있다.

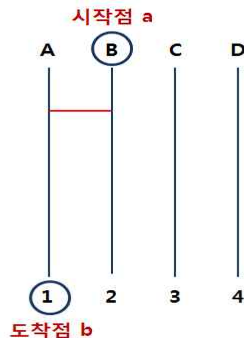


〈문제에 제시된 사다리 상태〉

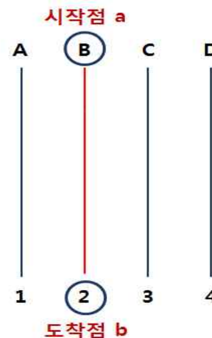
가로선과 세로선의 상태에 따른 2차원 점화식을 세울 수 있다.

DT[i][j] = i번째 가로선까지 진행했을 때 세로선 j까지 도착하는 최소 비용

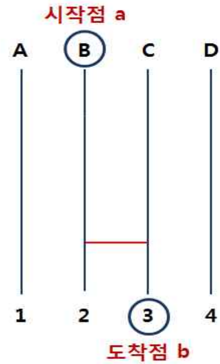
1. 먼저 DT[m][n]의 값을 최댓값으로 초기화한다.(∵최소 비용을 구하기 때문)
2. 가로선①을 지나기 전에 DT[0][j]를 초기화한다. 시작점이 2이고 2에서 각 세로선까지 가려면 가로선들을 추가해야 한다. 따라서 추가해야 할 가로선의 개수만큼 추가비용 Y를 곱해주면 비용을 구할 수 있다.



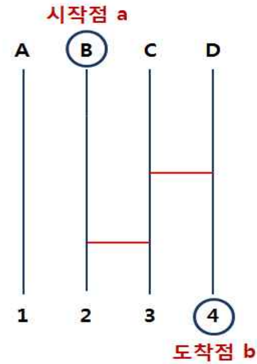
DT[0][1] = Y = 5



DT[0][2] = 0



DT[0][3] = Y = 5

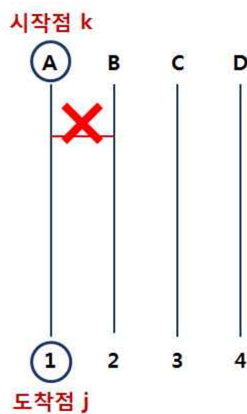


DT[0][4] = 2Y = 10

3. 본격적으로 DT[i][j]를 구한다. 입력으로 들어온 i번째 가로선까지 진행했을 때 세로선 k에서 세로선 j로 가는 최소비용을 갱신한다. i번째 가로선을 처리하기 위한 반복문이 필요하고, 이 반복문 내부에는 출발 위치 k에서 도착 위치 j로 가는 모든 경우를 탐색해야 하므로 3중첩 반복문이 필요하다. m, n의 제한이 크지 않아 3중첩 반복문을 사용하여도 시간 초과에 걸리지 않는다.

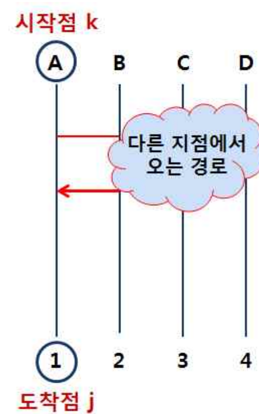
배열의 값을 갱신할 때 다음과 같은 3가지 경우로 나누어 처리한다. 다음 그림에 나와 있는 경우는 여러 가지 상황을 보여주기 위한 것이므로 문제의 예시와는 무관하다.

시작점 k와 도착점 j가 같은 경우(일직선인 경우)



세로선이 일직선상인 경우 가로선 제거 비용

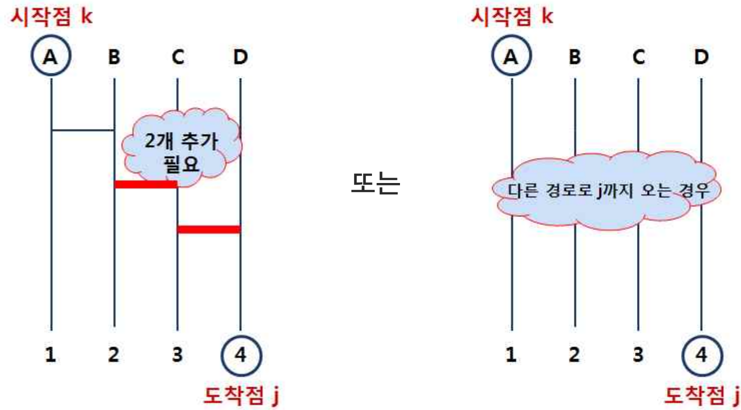
또는



다른 곳을 거쳐서오는 비용

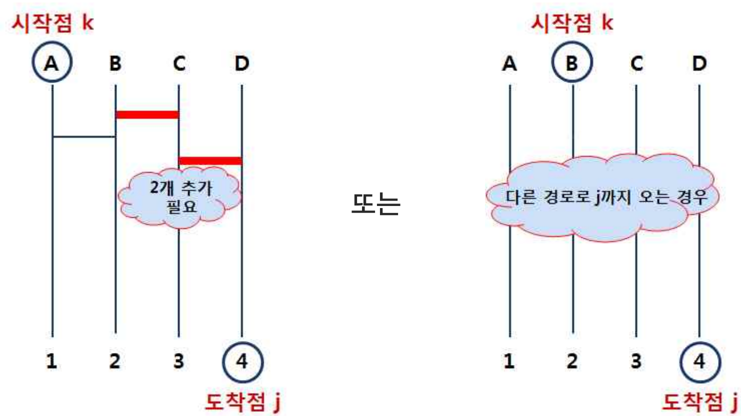
〈둘 중 비용이 적게 드는 것을 선택하여 갱신〉

시작점 k와 도착점 j 사이에 i번째 가로선이 존재하는 경우



〈둘 중 비용이 적게 드는 것을 선택하여 갱신〉

시작점 k와 도착점 j사이에 i번째 가로선이 존재하지 않는 경우



〈둘 중 비용이 적게 드는 것을 선택하여 갱신〉

4. m번째 가로선까지 진행하고 나면 DT[m][b]의 값도 3번 과정에 의해 구할 수 있다. 최종적으로 DT[m][b]를 출력한다.

이 점화식을 적용한 코드는 다음과 같다.

```
#include <stdio.h>
#include <stdlib.h>

int n, m;
int p[501];
int DT[501][101];
int isIn(int a, int b, int k)
{
    return ((a<=k && k<b) || (b<=k && k<a)) ? 1:0;
}

int main()
{
  int a, b, X, Y;
  int i, j, k;

  scanf("%d %d", &n, &m);
  scanf("%d %d %d %d", &a, &b, &X, &Y);
  for(i=1; i<=m; i++)
    scanf("%d", &p[i]);

  for(i=0; i<=m; i++)
    for(j=0; j<=n; j++)
      DT[i][j]=99999999;

  for(j=1; j<=n; j++)
    DT[0][j]=abs(j-a)*Y;

  for(i=1; i<=m; i++)
   for(j=1; j<=n; j++)
    for(k=1; k<=n; k++)
     if(j==k && (p[i]==k || p[i]+1==k))
        DT[i][j]=(DT[i-1][k]+X<DT[i][j])?DT[i-1][k]+X:DT[i][j];
     else if(isIn(j,k,p[i]))
        DT[i][j]= DT[i-1][k]+(abs(j-k)-1)*Y<DT[i][j]?
                  DT[i-1][k]+(abs(j-k)-1)*Y:DT[i][j];
     else
        DT[i][j]=DT[i-1][k]+abs(j-k)*Y<DT[i][j]?
                  DT[i-1][k]+abs(j-k)*Y:DT[i][j];
  printf("%d\n", DT[m][b]);
}
```

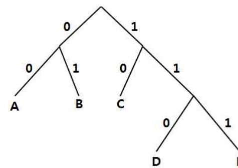
입력의 정의역의 최대 크기에 따라 실제 계산량을 구해보면, $500 \times 100 \times 100 = 5{,}000{,}000$이다. 따라서 제한 시간 안에 충분히 알고리즘을 수행할 수 있다. 이 알고리즘의 전체적인 계산량은 $O(mn^2)$으로 나타낼 수 있다.

문제 8

허프만 인코딩

허프만 인코딩은 탁월한 글자 압축 알고리즘이다. 이 알고리즘은 1952년 Davide Huffman에 의해 개발되었다.

기본 아이디어는 각 문자를 2진 순서를 이용해서 자주 등장하는 글자에게 비트를 작게 할당하고 아주 드문 글자에게 많은 비트를 할당하여 줄이는 방식이다. 그리고 모든 글자의 2진 순서는 다르다. 보통 이진트리형태로 표현하여 글자를 단말노드(트리의 마지막에 배치하는 형태이다. 예를 들어 아래의 그림과 같은 이진트리에 각 문자가 배치되었다고 하면



A 는 00, B는 01, C는 10, D는 110, E는 111로 코드를 갖게 된다. 이를 바탕으로 주어진 문자를 압축을 하게 된다. 여러분들은 역으로 허프만 코드가 주어질 경우 압축된 데이터를 원래의 문자로 변경하는 것이다. 주어진 압축 이진코드를 읽어 원래의 문자로 출력하는 것이다.

입력

첫줄에 정수 k(1<=k<=20)가 입력된다. 이것은 허프만 코드의 각 문자의 개수를 나타낸다.

다음 줄부터 k개의 문자와 그에 상응하는 이진코드 값이 공백으로 구분하여 입력된다. (최대 20자) 이며 문자는 A~Z, a~z 중에 선택된다. 그 다음 줄에 250자리 이하의 코드가 입력된다.

출력

첫줄에 코드를 원래의 문자로 출력한다.

입력 예	출력 예
5 A 00 B 01 C 10 D 110 E 111 00000101111	AABBE

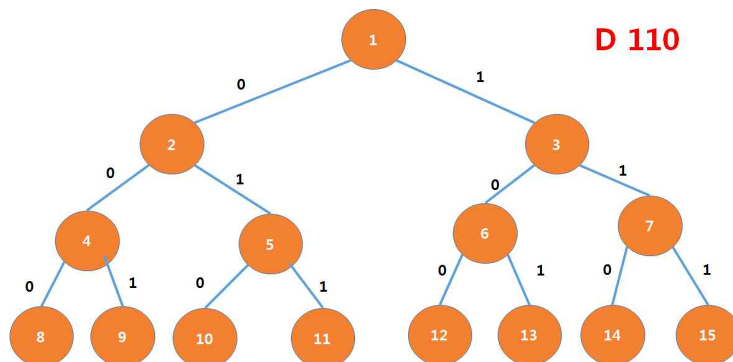
풀이

먼저 입력된 값을 바탕으로 트리를 완성하는 것이다. 주어진 범위의 k가 최대 20개 이므로 트리의 깊이가 최대 20이 될 수 있다. 이를 바탕으로 입력으로 들어오는 값들을 가지고 허프만 트리를 먼저 생성하는 것이다.

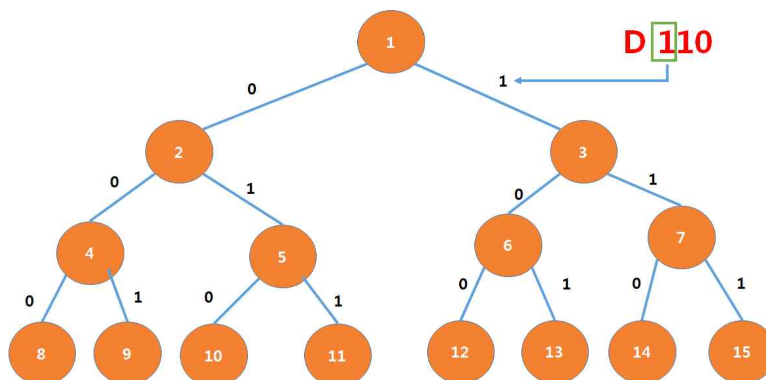
만드는 방법은 0이면 왼쪽자식이고 1이면 오른쪽 자식으로 간 다음 이 과정을 반복하다 마지막이면 해당위치에 문자값을 저장한다.

포화 이진트리를 통해 구현하면 그림과 같은 초기 상태에서 D 110인 허프만 코드를 허프만 트리로 만들어 보면

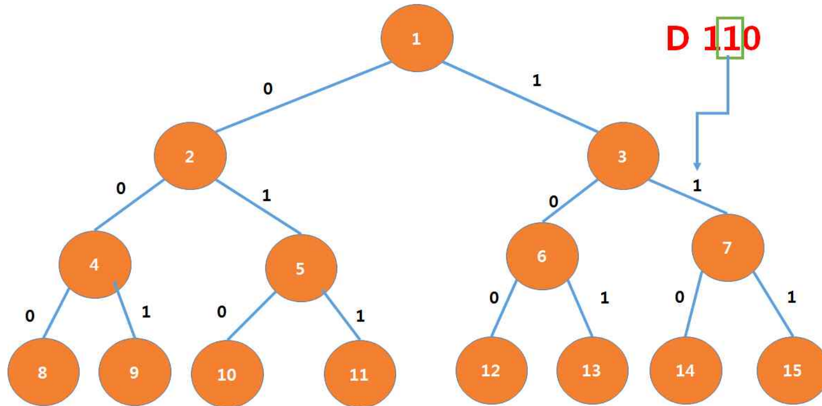
① 초기상태



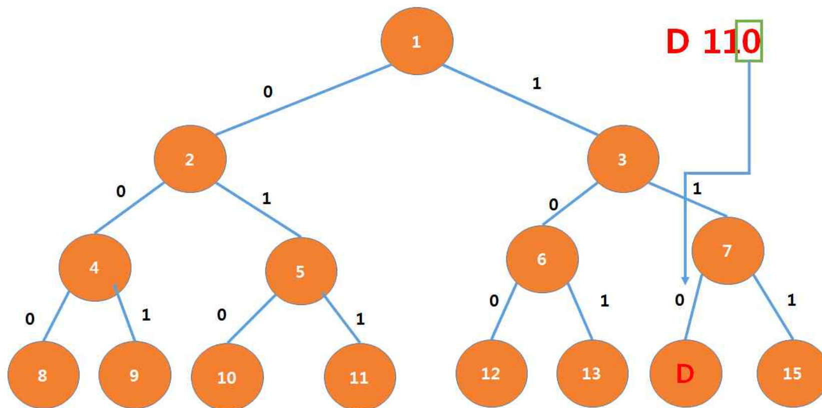
② 첫 문자 ‘1’



③ 두 번째 문자 '1'



④ 세 번째 문자 '0'



위와 같이 각 문자에 해당하는 허프만 코드를 허프만 트리로 완성한다.

str : 입력받은 문자
strvalue[] : 입력받은 허프만 코드
idx : 트리의 위치
tree[] : 만들고자 하는 허프만 트리
strvalue[idx] == '0' 이면 왼쪽자식으로 이동 idx*2
strvalue[idx] == '1' 이면 오른쪽 자식으로 이동 idx*2+1
strvalue[idx] == '₩0' 끝이면 트리에 해당 문자를 저장한다. tree[idx] = str

```
int k;
scanf("%d",&k);

for(int i=1; i<=k; i++)
{
  scanf("\n%c %s",&str,strvalue);

  int idx=1;

  for(int idx=0; strvalue[idx]!='\0'; idx++)
  {
    if(strvalue[idx]=='0') idx=idx*2;
    else idx=idx*2+1;
  }

  tree[idx]=str;
}
```

이를 바탕으로 입력된 문자열을 가지고 트리를 탐색해서 출력하면 된다. 소스코드는 다음과 같다.

```
#include <stdio.h>
char str, strvalue[30], data[300], tree[1500000];
int main()
{
  int k;
  scanf("%d", &k);

  for(int i=1; i<=k; i++)
  {
    scanf("\n%c %s", &str, strvalue);

    int idx=1;

    for(int idx=0; strvalue[idx]!='\0'; idx++)
    {
      if(strvalue[idx]=='0') idx=idx*2;
      else idx=idx*2+1;
    }
```

```
    tree[idx]=str;
  }

  scanf("\n%s", data);
  int idx=1;

  for(int i=0; data[i]!='\0'; i++)
  {
    if(data[i]=='0') idx=2*idx;
    else idx=2*idx+1;
    if(tree[idx]!=0)
    {
      printf("%c",tree[idx]);
      idx=1;
    }
  }

  return 0;
}
```

이 소스코드의 계산량은 문자의 개수가 k이고 문자의 이진코드 길이가 n이 라면 $O(kn)$이다.

위의 알고리즘은 상당히 많은 양의 메모리를 사용하고 있다. 이때 STL의 자료구조인 map[3])를 이용하면 쉽게 이를 해결할 수 있다. 즉, key와 value를 하나의 쌍으로 저장한 후 해당 key를 찾아서 value를 바로 출력하게 만드는 것이다.

---

3)std::map : set, map 등은 이진탐색트리의 구조로 삽입, 탐색 등을 $O(\lg n)$에 처리할 수 있는 효율적인 구조이다. (http://http://en.wikipedia.org/wiki/Associative_containers)

코드

```
#include <map>
#include <cstdio>
#include <string>
#include <iostream>

using namespace std;

int main()
{
  map<string, char> mp;
  int k, i;
  char ch;
  string str, a;

  cin >> k;

  for(i=0; i<k; i++)
  {
    cin >> ch >> str;
    mp[str]=ch;
  }

  cin >> str;

  for(i=0; i<str.length(); i++)
  {
    a+=str[i];
    if(mp[a])
    {
      cout << mp[a];
      a.clear();
    }
  }
  return 0;
}
```

참고

** cin, cout은 std::cin, std::cout c++에서 사용되는 console input, console output으로, 일반 C에서 scanf( ), printf( )와 비슷한 기능을 수행하지만 객체지향적인 입출력 방법으로 속도는 느리지만 더 많은 기능들을 제공한다.

이 소스코드의 계산량은 문자의 개수 k와 map을 만드는 시간 $O(\lg n)$이 필요하므로 $O(k^* \lg n)$이다.

문제 9

색상환

색을 표현하는 기본 요소를 이용하여 표시할 수 있는 모든 색 중에서 대표적인 색을 고리 모양으로 연결하여 나타낸 것을 색상환이라고 한다. 미국의 화가 먼셀(Munsell)이 교육용으로 고안한 20색상환이 널리 알려져 있다. 아래 그림은 먼셀의 20색상환을 보여준다.



「먼셀의 20색상환」

색상환에서 인접한 두 색은 비슷하여 언뜻 보면 구별하기 어렵다. 위 그림의 20색상환에서 다홍은 빨강과 인접하고 또 주황과도 인접하다. 풀색은 연두, 녹색과 인접하다. 시각적 대비 효과를 얻기 위하여 인접한 두 색을 동시에 사용하지 않기로 한다.

주어진 색상환에서 시각적 대비 효과를 얻기 위하여 서로 이웃하지 않은 색들을 선택하는 경우의 수를 생각해 보자.

먼셀의 20색상환에서 시각적 대비 효과를 얻을 수 있게 10개의 색을 선택하는 경우의 수는 2이지만, 시각적 대비 효과를 얻을 수 있게 11개 이상의 색을 선택할 수 없으므로 이 경우의 수는 0이다.

색상환 (계속)

주어진 정수 n과 k에 대하여, n개의 색으로 구성되어 있는 색상환(n색상환)에서 어떤 인접한 두 색도 동시에 선택하지 않으면서 서로 다른 k개의 색을 선택하는 경우의 수를 구하는 프로그램을 작성하시오.

입력

입력 파일의 첫째 줄에 색상환에 포함된 색의 개수를 나타내는 양의 정수 n이 주어지고, 둘째 줄에 n색상환에서 선택할 색의 개수 k가 주어진다.
(4<=n<=1,000), (1<=k<=n)

출력

첫째 줄에 n색상환에서 어떤 인접한 두 색도 동시에 선택하지 않고 k개의 색을 고를 수 있는 경우의 수를 1,000,000,003 (10억 3)으로 나눈 나머지를 출력한다.

입력 예	출력 예
4 2	2

출처: 한국정보올림피아드(2010 지역본선 중고등부)

풀이

먼저 이 문제를 재귀함수를 이용하여 하향식으로 설계해보자. 먼저 문제의 관계를 파악하기 위해서 문제를 잘 분석해야 한다.

이 문제는 원형으로 배치된 n개의 색상들 가운데 k개를 인접하지 않도록 고르는 것이 목적이다. 이 문제를 해결하기 위하여 다음과 같은 함수를 정의한다.

$f(n, k)$ = "n개의 색상 중 인접하지 않도록 k개의 색상을 고르는 경우의 수"

이와 같은 정의를 내릴 때, 다음과 같이 직접 구할 수 있는 값들을 정의할 수 있다.

$$f(n, k) = 0 \quad (k > \frac{n}{2}), \quad f(n, k) = n \ (k=1)$$

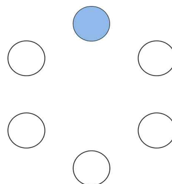
다음으로 문제의 관계를 파악해야 한다. 이 문제에서의 어려움은 처음 색상을 고르면 마지막 색상을 고를 수 없다는 점이다. 따라서 설계할 때 난이도가 다른 문제에 비해서 높다.

먼저 가장 첫 번째 색상에 대해서만 관찰을 시작하자. 모든 경우의 수를 나열 했을 때, 맨 첫 번째 색상만을 본다면 다음 2가지 경우만 존재할 수 있다.

① 가장 첫 번째 색상을 골라서 n개의 색상 중 k개의 색상을 인접하지 않게 고른 경우.
② 가장 첫 번째 색상을 고르지 않고 n개의 색상 중 k개의 색상을 인접하지 않게 고른 경우.

위의 ①의 경우 수와 ②의 경우의 수를 합한 결과가 전체의 해가 됨이 명확하다. 따라서 각 경우의 수를 정확하게 f 함수를 이용하여 구하면 된다.

먼저 ①의 문제를 f 함수로 표현해보자. 첫 번째 색상을 골랐으므로, $k-1$개의 색상을 더 고를 수 있다. 다음 그림은 6개의 색상이 있는 경우 중 첫 번째 색상을 고른 경우를 나타낸다.



이제 남은 색상들 중에 $k-1$가지의 색상을 더 골라야 한다. 일단 한 가지 색상을 선택했으므로 남은 색상은 $n-1$개다. 따라서 쉽게 생각하면 ①의 경우의 수를 다음과 같이 생각할 수 있다.

$f(n-1,\ k-1)$ 가지

하지만 잘 생각해보면 위의 경우의 수가 아니라는 사실을 알 수 있다. 왜냐하면 f 함수라는 함수 자체가 원형으로 배치된 상태에서 인접하지 않도록 고르는 경우의 수이다. 하지만 이 경우에는 첫 번째 색상을 골랐기 때문에 2번째 색과 n번째 색을 모두 고를 수 없다. 따라서 위와 같이 정의하면 잘못된 경우가 발생할 수 있다.

따라서 위와 같이 ①, ②를 정하면 해를 구하기 위한 관계를 정리하기 쉽지 않음을 알 수 있다. 물론 포함배제 등을 잘 해서 구할 수도 있지만 이보다 훨씬 간단한 아이디어로 접근하는 방법이 있다.

이 문제는 인접한 색을 연속으로 고를 수 없기 때문에 임의의 연속된 두 색을 골랐을 때, 두 색을 모두 고른 경우는 없다. 왜냐하면 문제에서 인접한 색을 고르지 못한다고 했기 때문이다.

따라서 첫 번째 색과 두 번째 색을 모두 고려하는 방법을 생각해보자. 이 경우에는 다음 2가지 경우가 있다. 이 두 가지 경우를 다음과 같이 정의하여 관계를 설계해 보자.

① 첫 번째나 두 번째 색 중 한 가지 색을 이미 고른 경우.
② 첫 번째 색과 두 번째 색 모두 고르지 않았을 경우.

임의의 두 색을 선택했을 때 위 두 가지 경우 이외의 경우는 없다. 따라서 ①의 경우의 수와 ②의 경우의 수를 모두 구하여 더하면 이 문제의 해가 된다.

먼저 ①의 경우의 수를 구해보자. 두 가지 색상 중 한 가지를 골랐으므로 나머지 한 가지는 안 고른 경우이다. 따라서 $k-1$가지 색을 더 고를 수 있다.

그리고 남아 있는 색상이 $n-2$가지이며 이 때 원형으로 배치되었다고 생각해야 된다. 왜냐하면, 빠져 있는 2개의 색 중 하나를 이미 골랐기 때문에 절대로 $n-2$개 중 맨 끝의 2개를 동시에 고를 수 없다. 따라서 $n-2$개로 구성된 원형 색상환으로 봐도 무방하므로

다음과 같은 식으로 구할 수 있다.

$f(n-2,\ k-1)$ 가지

이 경우가 ①의 경우의 수다. 다음으로 ②의 경우의 수에 대해서 생각해보자.

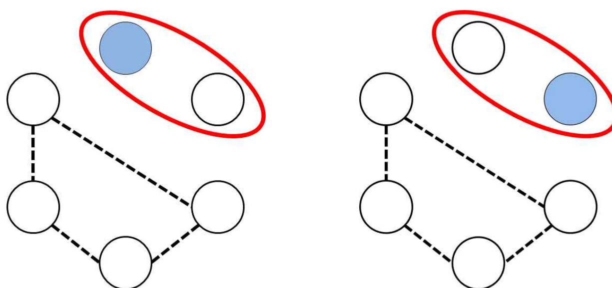
이 경우는 첫 번째, 두 번째 색을 모두 고르지 않았기 때문에 k가지 색을 더 골라야 하며, 마지막 두 색을 연속으로 고를 수 있는 문제가 된다.

만약 이 경우 단순히 $f(n-2,\ k)$라고 하면 문제의 정의에 의해서 처음 색과 마지막 색을 고를 수 없다. 하지만 실제로는 두 색을 고를 수 있는 상태이다. 따라서 사용하지 않은 2개의 색상 중 하나를 더 추가한 $n-1$개의 원소를 가지는 색상환이라고 가정하면 처음과 끝의 원소를 다 고를 수 있다. 따라서 두 번째 경우의 수는 다음과 같이 나타낼 수 있다.

$f(n-1,\ k)$ 가지

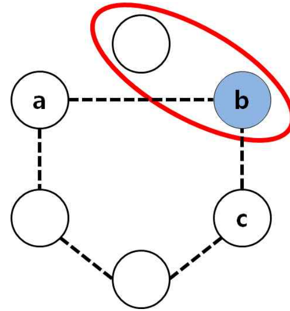
위의 2가지 경우를 그림으로 나타내면 다음과 같다.

①의 경우의 수를 구할 때 n개 중 원으로 칠한 2개를 제외한 $n-2$개를 새로운 색상환으로 생각하고 이 중 $k-1$개를 고른다. 어떻게 골라도 아래 두 그림 중 하나에 속하므로 원에 있는 둘 중 하나를 칠한 경우의 수가 된다.



②의 경우의 수는 원에 속한 2개를 모두 고르지 않으므로 k개를 더 골라야 한다. 만약 원에 속한 2개를 제외한 $n-2$개에서 k개를 고른다면 a와 c를 동시에 고를 수 없다. 하지만 이 경우는 실제로 a와 c를 동시에 고를 수 있어야 한다. 따라서 b를 대상에 추가하여

b를 무조건 고르지 않도록 생각하면 a와 c를 동시에 고를 수 있으므로 $n-1$개의 색상환에서 k개를 고르는 경우의 수와 같다.



따라서 다음과 같은 관계로 정리할 수 있다.

$$f(n,\ k) = \begin{cases} 0 & (k > \frac{n}{2}) \\ n & (k = 1) \\ f(n-2,\ k-1) + f(n-1,\ k) & (otherwise) \end{cases}$$

이 식을 재귀함수로 작성한 소스코드는 다음과 같다.

```
#include <cstdio>
#define MOD 1000000003

int f(int n, int k)
{
  if(k>n/2) return 0;
  else if(k==1) return n;
  else return (f(n-2,k-1)+f(n-1,k))%MOD;
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  printf("%d\n", f(n,k));
  return 0;
}
```

이 소스코드는 답은 올바르게 출력하나 n이 증가함에 따라서 중복호출의 수가 기하급수적으로 늘어나게 되므로 시간이 너무 오래 걸려 제한된 시간 내에 해를 출력할 수 없다. 따라서 중복호출을 막기 위하여 동적표를 활용한 메모이제이션 기법으로 해결할 수 있다.

동적표는 다음과 같이 정의한다.

$$DT[n][k] \ = f(n,\ k) \text{ 를 기록}$$

동적표를 적용한 결과는 다음과 같다.

```
#include <cstdio>
#define MOD 1000000003

int DT[1001][1001];

int f(int n, int k)
{
  if(k>n/2) DT[n][k]=0;
  else if(k==1) DT[n][k]=n;
  else
  {
    if(!DT[n][k])
      DT[n][k]=(f(n-2,k-1)+f(n-1,k))%MOD;
  }
  return DT[n][k];
}

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  printf("%d\n", f(n,k));
  return 0;
}
```

이번에는 같은 관계식을 이용하여 상향식으로 작성해 볼 수 있다. 관계식은 다음과 같이 정의한다.

$$DT[n][k] = \begin{cases} 0 & (k > \frac{n}{2}) \\ n & (k = 1) \\ DT[n-2][k-1] + DT[n-1][k] & (otherwise) \end{cases}$$

이 관계식을 이용한 소스코드는 다음과 같다.

```
#include <cstdio>
#define MOD 1000000003

int DT[1001][1001];

int main()
{
  int n, k;
  scanf("%d %d", &n, &k);
  for(int i=2; i<=n; i++)
    for(int j=1; j<=n/2; j++)
      if(j==1) DT[i][j]=i;
      else DT[i][j]=(DT[i-2][j-1]+DT[i-1][j])%MOD;
  printf("%d\n", DT[n][k]);
  return 0;
}
```

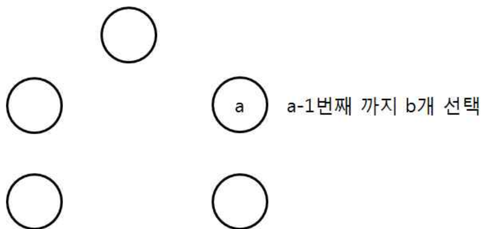
이와 같이 점화식만 잘 세우면 상향식으로 깔끔하게 코딩할 수 있다. 계산량은 메모이제이션으로 작성한 것과 같지만, 실제 속도는 반복문을 이용한 상향식이 더 빠르다. 따라서 점화식을 세우기 쉬운 문제들은 상향식으로 접근하는 것이 가장 좋은 방법이라고 할 수 있다.

하지만 이 문제와 같이 관계식을 만들기 쉽지 않거나, 아이디어가 잘 떠오르지 않는 경우에도 문제를 해결할 수 있다. 이 때 활용할 수 있는 방법이 전체탐색법을 기반으로 한 다이나미컬 백트래킹 방법이다.

먼저 탐색기반 설계로 이 문제를 접근해보자. 먼저 첫 번째 색상부터 탐색을 시작해야 한다. 다음으로 현재 색상을 선택할 것인지 선택하지 않을 것인지를 이용하여 탐색을 진행한다. 따라서 탐색하는 함수는 다음과 같이 정의한다.

$f(a,\ b)\ =$
"현재 a번째 색상에 대한 검색할 차례이며, b개의 색상을 인접하지 않도록 선택한 상태"

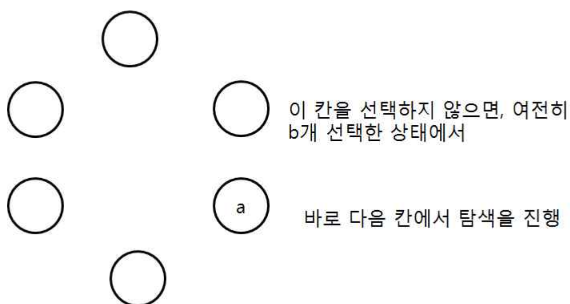
이 정의대로 탐색을 진행하기 위해서는 다음 그림과 같은 선택을 할 수 있다.



이 상태에서 선택 가능한 상태는 이 칸을 선택할 경우①, 선택하지 않을 경우②가 된다.

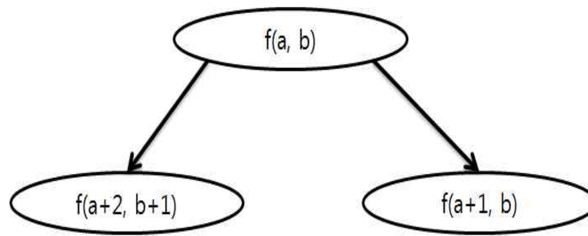


① 한 칸을 선택하면 다음 칸은 선택할 수 없음이 명확하다!! 따라서 그림과 같다.



② 현재 칸을 선택하지 않으면 바로 다음 칸을 선택할 수 있기 때문에, 그림과 같다.

이 과정을 함수의 형태로 표현하면 다음과 같다.



그리고 처음에 출발할 때는 다음 2가지 상태가 있다. 처음 색깔을 선택하는 경우와 그렇지 않은 경우, 즉 다음과 같이 출발할 수 있다.

$f(1,\ 0)$: "첫 번째(0번) 색상을 선택하지 않고 1번 색으로 이동, 따라서 현재 선택은 0"
$f(2,\ 1)$: "첫 번째(0번) 색상을 선택하고 2번 색으로 이동, 따라서 현재 선택은 1"

이와 같이 선택할 경우에는 해가 틀린다. 왜냐하면 배치가 선형이 아니기 때문이다. 따라서 처음 색상을 선택했는지의 여부를 알 수 없다. 만약 처음 색상을 선택했으면, 마지막 색을 선택할 수 없다.

따라서 여기에 처음 색상의 선택여부를 bool 변수인 can에 true, false로 설정하여 이 값이 true면 마지막 색을 선택할 수 있고, false면 마지막 색을 선택하지 못하도록 설정하자. 탐색기반 설계는 설계자가 정하는 어떤 값이든 활용 가능하다. 따라서 탐색함수는 다음과 같이 된다.

$f(1,\ 0, true)$: "첫 번째(0번) 색상을 선택하지 않고 1번 색으로 이동, 따라서 현재 선택은 0"
$f(2,\ 1, false)$: "첫 번째(0번) 색상을 선택하고 2번 색으로 이동, 따라서 현재 선택은 1"

이 함수를 이용한 소스코드는 다음과 같다.

```
#include <cstdio>
#define MOD 1000000003

int n, k, ans;

void f(int a, int b, bool can)
{
  if(a>=n || b==k)
  {
    ans+=((int)((b==k) && (a<=n || can));
    ans%=MOD;
    return;
  }
  f(a+1, b, can);
  f(a+2, b+1, can);
}

int main()
{
  scanf("%d %d", &n, &k);
  f(1, 0, true);
  f(2, 1, false);
  printf("%d\n", ans);
  return 0;
}
```

하지만 이 방법은 중복호출이 발생하므로 매우 느리다. 따라서 동적표를 적용한 다이나미컬 백트래킹으로 작성해보자. 먼저 void 함수를 정수형으로 바꾸어야 한다.

적용한 결과는 다음과 같다.

줄	코드	참고

```
1  #include <cstdio>
2  #include <memory.h>
3  #define MOD 1000000003
4  
5  int n, k;
6  int DT[1001][1001][2];
7  
8  int f(int a, int b, bool can)
9  {
10   if(a>=n || b==k)
11     return (int)((b==k) && (a<=n || can));
12   else if(DT[a][b][can]==-1)
13     DT[a][b][can]=(f(a+1,b,can)+f(a+2,b+1,can))%MOD;
14   return DT[a][b][can];
15 }
16 
17 int main()
18 {
19   memset(DT, -1, sizeof(DT));
20   scanf("%d %d", &n, &k);
21   printf("%d\n", (f(1,0,true)+f(2,1,false))%MOD);
22   return 0;
23 }
```

이와 같이 다이나미컬 백트래킹은 매우 다양하게 활용할 수 있다. 특히 이 소스에서 활용한 변수 can의 의미는 알고리즘의 설계를 매우 유동적으로 할 수 있도록 한다. 따라서 수학적으로 아이디어가 부족하더라도 충분히 문제를 해결할 수 있는 방법을 제공한다.

이 소스에서 주의할 점은 19행에서 DT를 −1로 초기화 한 점이다. 실제 해 중에 0인 해가 존재하므로 0을 메모하지 않은 것으로 판단하면 중복호출을 줄이는 효과가 떨어진다. 따라서 −1로 초기화를 하면 이런 점을 방지할 수 있다.

그리고 출력할 때 두 함수를 따로 호출하므로 이 때는 MOD가 적용되지 않을 가능성도 있으므로 잘 파악해두기 바란다.

사실 조금 더 엄밀하게 수학적으로 접근하면 can을 사용하지 않고 다음과 같이 활용할 수도 있다. 이 소스가 왜 성립하는지는 스스로 생각해 보기 바란다.

```
#include <cstdio>
#include <memory.h>
#define MOD 1000000003

int n, k;
int DT[1001][1001];

int f(int a, int b)
{
  if(a>=n || b==k)
    return (int)(b==k);
  else if(DT[a][b]==-1)
    DT[a][b]=(f(a+1,b)+f(a+2,b+1))%MOD;
  return DT[a][b];
}

int main()
{
  memset(DT, -1, sizeof(DT));
  scanf("%d %d", &n, &k);
  printf("%d\n", (f(1,0)+f(3,1))%MOD);
  return 0;
}
```

사람에 따라 다르지만 관계식이 잘 나오지 않을 때에는 전체탐색으로부터 시작하는 다이나믹컬 백트래킹도 매우 쉽게 알고리즘을 설계할 수 있는 설계방법 중 하나이므로 꼭 익히도록 하자.

문제 10

maximum sum(L)

n개의 원소로 이루어진 집합이 있다. 이 집합에서 최대로 가능한 부분합을 구하는 것이 문제이다.

부분합이란 n개의 원소 중 i번째 원소로부터 j번째 원소까지의 연속적인 합을 의미한다. (단, 1 <= i <= j <= n) 만약 다음과 같이 6개의 원소로 이루어진 집합이 있다고 가정하자.

6 -7 3 -1 5 2

이 집합에서 만들어지는 부분합 중 최댓값은 3번째 원소부터 6번째 원소까지의 합인 9이다.

입력

첫 줄에 원소의 수를 의미하는 정수 n이 입력되고, 둘째 줄에 n개의 정수가 공백으로 구분되어 입력된다.
(단, 2 <= n <= 100,000; 각 원소의 크기는 -1000에서 1000 사이의 정수이다.)

출력

입력된 그래프가 두 색으로 칠할 수 있는 그래프인지 판단하고 아래 예에 나온 형식에 맞게 결과를 출력하라.

입력 예	출력 예
6 6 -7 3 -1 5 2	9

풀이

이 문제는 중급편에서 $O(n^3)$으로 3차원 구조의 선형탐색으로 해결했었던 문제이다. 이번에는 이 문제의 입력값 n이 100,000으로 매우 크다. 따라서 다른 방법을 적용해야 한다. 먼저 하향식으로 재귀함수를 이용하여 설계해보자.

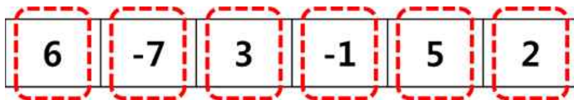
이 문제를 처음 접할 때, 관계설정을 하기 위한 함수 f를 다음과 같이 설정하는 학생들이 많다.

$f(n)$ = "처음부터 n까지의 원소들로부터 구할 수 있는 최대 부분합"

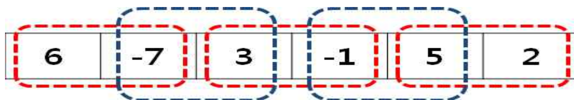
이와 같이 정의하면 가장 쉽게 해결할 수 있을 것처럼 보인다. 하지만 이렇게 설계할 경우에 $f(n)$을 이용하여 다음 $f(n+1)$을 구할 방법이 없다. 왜냐하면 $f(n)$에서의 최대 부분합이 어떤 원소로부터 시작하여 어떤 원소로 끝나는지에 대한 정보가 전혀 없기 때문이다.

따라서 $n+1$번째 원소를 연속적으로 연결할 수 있을지 여부도 불투명하기 때문에 진행하기 어렵다. 다시 한 번 이 문제를 해결하기 위한 관계를 찾을 수 있는 아이디어를 잘 생각해보자.

주어진 입력값은 다음과 같다.

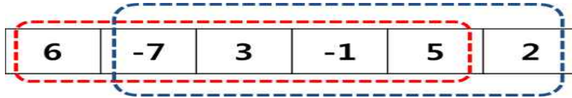


먼저 그림과 같이 길이가 1인 부분합이 모두 6가지가 있다.



다음으로 길이가 2인 부분합이 5가지가 있다.

:
:



길이가 5인 부분합이 그림과 같이 2개 있고 마지막으로 길이가 6인 부분합은 1개가 있을 것이다.

위 그림과 같이 모든 부분합을 일일이 그려보아도 구간합의 시작 원소, 혹은 마지막 원소는 n개를 넘을 수 없다는 사실을 알 수 있다. 즉, 어떤 구간이더라도 반드시 주어진 n개의 원소 중 하나로 시작하거나 끝난다는 사실이 중요하다.

여기서는 하향식으로 설계할 것이므로 끝나는 원소에 주목하자. 따라서 함수를 다음과 같이 다시 정의해보자.

$f(n)$ = "n번째 원소를 마지막 원소로 하는 최대 부분합"

이 정의를 이용하면 $f(n)$을 $f(n-1)$로부터 구할 수 있다.

만약 $f(n-1)$이 k라면 $f(n)$은 반드시 n번째 원소를 마지막으로 해야하므로, n번째 원소는 반드시 포함한다. 그렇다면 선택지는 2가지이다. 앞의 원소와 연결할 것인지, 하지 않을 것인지만 선택하면 된다.

따라서 다음과 같은 관계를 유도할 수 있다.

$$f(n) = \max\{\ f(n-1) + S[n],\ S[n]\ \}$$

원리는 간단하다 n번째 원소로 끝나는 최대 부분합은 n번째 원소 하나로 끝나거나, 아니면 여러 어떤 임의의 원소 k $(1 \leq k < n)$로부터 시작하여 n까지 더하거나 한 값이다. 그러나 $f(n-1)$의 값은 정의에 의해서 이미 $n-1$로 끝나는 최대 부분합이므로 k를 따로 구할 필요 없이 바로 연결할 수 있다.

이 관계를 다시 정리하면

$$f(n) = \begin{cases} S[1] & (n=1) \\ \max\{\ f(n-1) + S[n],\ S[n]\ \} & (n>0) \end{cases}$$

그리고 이 문제는 앞선 문제들과 달리 구하는 해가 $f(n)$이 아니다. $f(n)$은 n번째 원소로 끝나는 최대 부분합일 뿐이다. 문제의 해는 전체 중 최대 부분합이므로 모든 f 함수에 대한 고려가 필요하다. 따라서 구하고자 하는 해는 다음과 같다.

$$answer = \max\{\ f(k) \mid 0 \leqq k < n\ \}$$

이 관계를 이용하여 구현한 하향식 재귀 알고리즘은 다음과 같다.

코드	참고

```
#include <cstdio>
#define INF 987654321

int S[100000], n, ans=-INF;

int max(int a, int b)
{
  return a>b ? a:b;
}

int f(int k)
{
  if(k==0) return S[0];
  else return max(f(k-1)+S[k], S[k]);
}

int main()
{
  scanf("%d", &n);
  for(int i=0 ; i<n; i++)
    scanf("%d", S+i);
  for(int i=0; i<n; i++)
    ans=max(ans, f(i));

  printf("%d\n", ans);
}
```

하지만 위 소스코드는 중복호출이 많이 발생되므로 제한된 시간에 모두 해결할 수 없다. 따라서 동적표를 이용하여 메모이제이션 기법을 활용해야 한다.

메모이제이션 기법을 활용한 소스코드는 다음과 같다.

```
#include <cstdio>
#define INF 987654321

int S[100000], n, ans=-INF;
int DT[100000];

int max(int a, int b)
{
  return a>b ? a:b;
}

int f(int k)
{
  if(k==0) return DT[k]=S[0];
  else if(!DT[k]) DT[k]=max(f(k-1)+S[k], S[k]);
  return DT[k];
}

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%d", S+i);
  for(int i=0; i<n; i++)
    ans=max(ans, f(i));

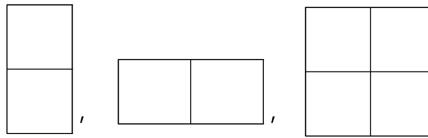
  printf("%d\n", ans);
}
```

이와 같이 재귀함수만 잘 작성할 수 있으면 동적표로의 적용은 매우 간단하다.

문제 11

타일채우기(L)

2*1 혹은 2*2크기의 타일을 2*n 크기의 직사각형모양 틀에 넣으려고 한다. 이 때 가능한 경우의 수를 구하여라.



경우의 수가 커지므로, 주어지는 수 m으로 나눈 나머지를 출력한다.

입력

첫 줄에는 직사각형 틀의 가로 길이 n이 주어진다.
둘째 줄에는 m이 주어진다. (1<=n<=100,000, 1<=m<=40,000)

출력

경우의 수를 m으로 나눈 나머지를 출력한다.

입력 예	출력 예
8 100	71
8 2	1

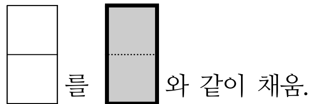
풀이

먼저 이 문제를 재귀함수를 이용하여 하향식으로 설계해보자. 앞에서도 설명했지만 재귀함수를 이용한 하향식 설계는 대부분 분할정복법으로 이루어지는 경우가 많다.

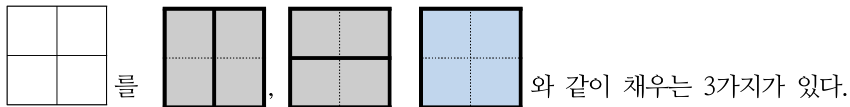
이 문제를 분할정복법으로 해결하기 위하여 먼저 바로 해결 가능한 사례와 그 나머지로 분할하여 설계할 수 있다. 바로 해결 가능한 사례는 문제의 크기 n이 1인 경우와 2인 경우이다.

그림과 같이 크기가 1인 사례는 도미노 하나를 세로로 채우는 경우가 있고 크기가 2인 사례는 가로로 도미노 2개를 채우는 방법과 세로로 도미노 2개를 채우는 방법, 마지막으로 2*2블록을 채우는 방법으로 모두 3가지 방법이 있다.

"2×1의 판을 채우는 경우도 1가지뿐이다."

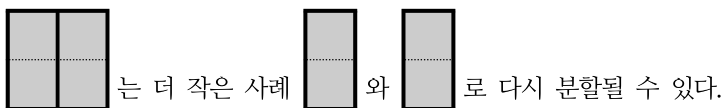


"2×2의 판을 채우는 경우는 3가지가 있다."



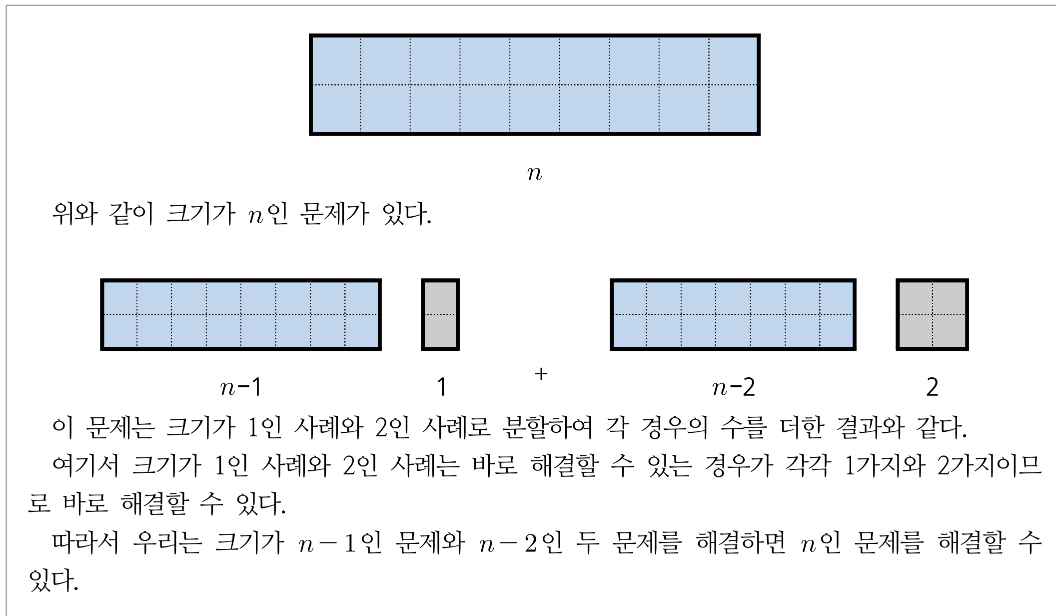
위 그림과 같이 바로 해결할 수 있는 사례를 정의할 때 문제점이 생긴다. 이는 분할정복으로 문제를 설계할 때 중요한 요소로 꼭 익혀둘 필요가 있는 부분이다.

바로 해결할 수 있는 사례란 더 이상 작은 사례로 분할할 수 없는 상황을 말한다. 그런데 사례가 2인 3가지 경우에서 가장 먼저 있는 그림은 다음과 같이 더 작은 사례 2개로 분할된다.



따라서 이 사례를 넣어서 식을 세울 경우 중복된 경우가 발생되므로 주의해야 한다. 그리고 크기가 3인 경우는 더 이상 분할될 수 없는 사례, 즉 바로 해결될 사례로 만들 수 없다. 따라서 바로 해결할 수 있는 사례는 크기가 1인 사례 1가지와 크기가 2인 사례 2가지로 볼 수 있다.

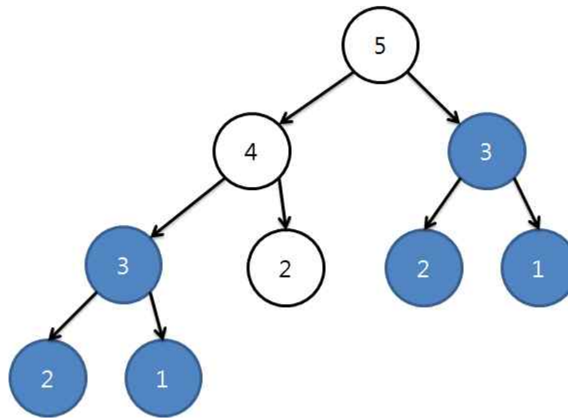
다음 그림은 크기가 1인 사례와 $n-1$인 사례로 분할하여 해결하는 과정을 나타낸다.



위와 같이 크기가 $n-1$인 문제와 $n-2$인 문제는 같은 방법으로 다시 분할할 수 있으므로 다음과 같은 점화식으로 문제를 해결할 수 있다.

$$f(n) = f(n-1) + 2f(n-2)$$

위 식은 이미 앞에서 다룬 식과 같다. 이 식을 재귀함수로 구현할 경우 다음 그림과 같이 엄청난 계산의 중복이 발생된다. 다음 그림은 $n=5$일 때, 호출되는 과정을 보여주는 트리 구조를 나타낸다.



위 그림과 같이 5가 호출될 때, 4와 3이 호출되고 4에서 다시 3이 호출된다. 3이하의 노드는 그림과 같이 동일하며 모두 중복호출이 이루어진다. 만약 n 의 값이 100 이라면 중복호출의 수는 얼마나 될까?

대략적으로 계산해보면 노드 하나가 2개의 노드를 호출하므로 대략적으로 2^{100} 개에 비례하는 만큼의 호출이 이루어질 것이다. 하지만 실제 노드에 적힌 수는 100가지뿐이다. 즉 서로 다른 노드 100개가 2^{100} 에 비례하는 횟수만큼 등장한다는 의미이다. 따라서 중복호출의 횟수는 상상을 초월한다.

따라서 동적표를 이용한 메모이제이션 기법을 이용하여 중복호출을 줄이면 선형시간에 해결할 수 있는 알고리즘을 만들 수 있다.

코드	참고

```
#include <cstdio>

int m;

int f(int n)
{
  if(n==1) return 1%m;
  else if(n==2) return 3%m;
  else return (f(n-1)+2*f(n-2))%m;
}

int main()
```

코드	참고

```
{
  int n;
  scanf("%d %d", &n, &m);
  printf("%d\n", f(n));
  return 0;
}
```

이 알고리즘은 동적표를 적용하지 않고 바로 정복할 수 있는 크기와 그 나머지 부분으로 분할하여 해결한 알고리즘으로 지수시간에 비례하는 실행시간을 가지는 알고리즘이다. 따라서 매우 느리게 동작한다.

코드	참고

```
#include <cstdio>

int DT[100001], m;

int f(int n)
{
  if(n==1) return DT[n]=1%m;
  else if(n==2) return DT[n]=3%m;
  else
  {
    if(DT[n]==0)
      DT[n]=(f(n-1)+2*f(n-2))%m;
    return DT[n];
  }
}

int main()
{
  int n;
  scanf("%d %d", &n, &m);
  printf("%d\n", f(n));
  return 0;
}
```

동적표를 적용하여 선형시간으로 낮춘 알고리즘이다. 성능은 비교할 수 없을 만큼 빠르다.

이번에는 위에서 만든 점화식을 반복문을 이용하여 상향식으로 작성해보자. 이는 재귀호출에 걸리는 시간을 줄일 수 있으므로 매우 효율적으로 동작한다. 상향식으로 해결한 알고리즘은 다음과 같다.

```
#include <cstdio>

int DT[100001];

int main()
{
  int n, m;
  scanf("%d %d", &n, &m);
  DT[1]=1%m, DT[2]=3%m;
  for(int i=3; i<=n; i++)
    DT[i]=(DT[i-1]+2*DT[i-2])%m;
  printf("%d\n", DT[n]);
  return 0;
}
```

이와 같이 선형시간에 해결할 수 있는 다양한 방법이 존재한다. 다음으로 새로운 분할정복방법에 대해서 생각해보자.

분할정복은 위에서 소개한 바와 같이 바로 정복할 수 있는 크기와 나머지로 분할하면 설계는 쉬우나, 효율은 매우 떨어진다. 가장 효율적으로 분할할 수 있는 방법은 무엇인가?

조금 생각해보면 크기가 같은 여러 개의 사례로 분할하는 것이 효율적이라는 사실을 금방 알 수 있다. 이 문제에서는 크기가 같은 2개의 사례로 나누면 쉽게 해결할 수 있다는 것을 이미 앞에서 다루었으며, 그 점화식은 다음과 같다.

$$f(n) = \begin{cases} 1 & (n \le 1) \\ f(\frac{n}{2})^2 + 2f(\frac{n}{2}-1)^2 & (n > 1,\ n\text{은 짝수}) \\ f(\left[\frac{n}{2}\right])f(\left[\frac{n}{2}\right]+1) + 2f(\left[\frac{n}{2}\right]-1)f(\left[\frac{n}{2}\right]) & (n > 1,\ n\text{은 홀수}) \end{cases}$$

위 점화식을 이용하여 재귀함수로 구현한 코드는 앞에서 소개했으므로, 이 코드를 동적표를 이용하여 중복호출을 배제한 알고리즘은 다음과 같다.

```
#include <cstdio>

int n, m, DT[100001];

long long int f(int k)
{
  if(k<=1) return DT[k]=1%m;
  else
  {
    if(DT[k]==0)
    {
      if(k%2) DT[k]=(f(k/2)*f(k/2+1)+2*f(k/2)*f(k/2-1))%m;
      else DT[k]=(f(k/2)*f(k/2)+2*f(k/2-1)*f(k/2-1))%m;
    }
    return DT[k];
  }
}

int main()
{
  scanf("%d %d", &n, &m);
  printf("%lld\n", f(n));
  return 0;
}
```

이 알고리즘은 앞의 알고리즘들과는 비교할 수 없을 만큼 빠른 속도로 동작한다. 이 알고리즘은 로그시간에 동작하는 알고리즘으로 n 의 값이 커지더라도 매우 효율적으로 값을 구할 수 있는 방법이다.

이와 같이 분할정복법을 효율적으로 작성하면 상향식 동적계획법보다 효율적인 알고리즘을 설계할 수도 있으므로 잘 익혀둘 필요가 있다.

문제 12

거스름돈(L)

여러분은 실력을 인정받아 전 세계적으로 사용할 수 있는 자동판매기용 프로그램의 개발을 의뢰받았다. 거스름돈에 사용될 동전의 수를 최소화하는 것이다.

입력으로 거슬러 줘야할 돈의 액수와 그 나라에서 이용하는 동전의 가짓수 그리고 동전의 종류가 입력되면 여러 가지 방법들 중 가장 적은 동전의 수를 구하는 프로그램을 작성하시오.

입력

첫 번째 줄에는 거슬러 줘야할 돈의 액수 m이 입력된다.
(10 <= m <= 10,000)
다음 줄에는 그 나라에서 사용되는 동전의 종류의 수 n이 입력된다.
(1 <= n <= 10)
마지막 줄에는 동전의 수만큼의 동전 액수가 오름차순으로 입력된다.
(10 <= 액수 <= m)

출력

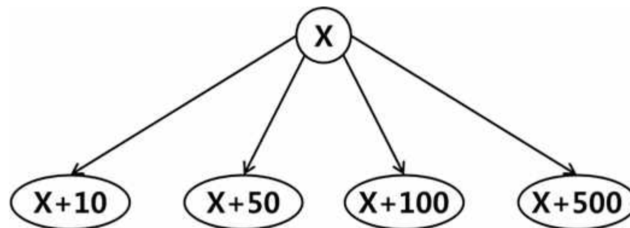
최소의 동전의 수를 출력한다.

입력 예	출력 예
730 5 10 50 100 500 1250	6

풀이

이 문제는 중급편에서 다룬 문제로 중급편에서 전체탐색법 및 탐색공간의 배제를 통하여 해결을 시도했으나 제한시간 내에 해결할 수 없었던 문제이다.

먼저 중급편에서 작성한 다음과 같은 탐색구조를 이용하여 문제를 해결해 보자. 구조화하는 방법은 다음 그림과 같다. 이 때 x 의 값은 지금까지 지불한 액수이며 사용 가능한 동전은 4가지 종류로 10원, 50원, 100원, 500원일 때를 가정한 것이다.



서로 다른 1개의 지불하는 방법에 대해서 탐색 상태를 정의

처음에 0원으로 시작하여 각 동전을 지불해 나가며, 지불할 금액과 일치할 때의 깊이가 지불한 동전의 개수이므로, 지불할 금액과 일치하는 최소 깊이를 구하는 구조로 다음과 같이 백트래킹 알고리즘을 작성할 수 있다.

코드	참고

```
#include <stdio.h>

int m, n, coin[10], ans = 987654321;

void solve(int mon, int d)
{
  if(mon>m) return;
  if(mon==m)
  {
    if(d<ans) ans=d;
    return;
  }
  for(int i=0; i<n; i++)
    solve(mon+coin[i], d+1);
}

```

```c
int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n; i++ )
    scanf("%d", coin+i);
  solve(0, 0);
  printf("%d\n", ans);
  return 0;
}
```

이 알고리즘에서 재귀함수 solve를 정수를 반환하는 함수 f로 다음과 같이 변환할 수 있다.

```c
#include <stdio.h>

int m, n, coin[10];

int min(int a, int b) {return a>b ? b:a;}

int f(int c)
{
  int ans=987654321;
  if(c==m) return 0;
  for(int i=0; i<n; i++)
    if(c+coin[i]<=m)
      ans=min(ans, f(c+coin[i])+1);
  return ans;
}

int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d", coin+i);
  printf("%d\n", f(0));
  return 0;
}
```

이와 같이 백트래킹 함수를 정수형 함수로 바꾸어도 여전히 매우 많은 시간이 걸리는 것은 같지만, 이와 같은 형태에서는 동적표를 적용하여 호출의 중복을 배제할 수 있다. 즉, 백트래킹 기반의 동적계획법을 구현할 수 있다.

$$DT[n] \;=\; f(n)\text{값을 기록}$$

이 알고리즘에 동적표를 적용한 결과는 다음과 같다.

코드	참고

```
#include <stdio.h>

int m, n, coin[10], DT[10001];

int min(int a, int b) {return a>b?b:a;}

int f(int c)
{
  if(c==m) return DT[c]=0;
  if(DT[c]==0)
  {
    DT[c]=987654321;
    for(int i=0; i<n; i++)
      if(c+coin[i]<=m)
        DT[c]=min(DT[c], f(c+coin[i])+1);
  }
  return DT[c];
}

int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d", coin+i);
  printf("%d\n", f(0));
  return 0;
}
```

이와 같이 구현하면 설계는 백트래킹과 같이 어렵지 않게 할 수 있으며, 효율은 동적계획법과 같은 효율이 나온다. 이 문제의 경우에는 선형시간으로 동작한다. 따라서 매우 효율적인 설계방법이라고 할 수 있다.

다음으로 상향식 동적계획법으로 이 문제를 해결해보자. 상향식 동적계획법으로 문제를 해결하기 위해서는 관계식을 정의해야 한다. 먼저 $DT[n]$을 다음과 같이 정의하자.

$$DT[n] \ = \ n\text{원을 지불하는데 필요한 최소 동전의 수}$$

위 정의에 대한 관계식을 만들 때, 앞에서 설계했던 탐색함수를 잘 이용하면 쉽게 관계를 유도할 수 있다. 결국은 n원을 지불하기 위해서 마지막에 사용할 수 있는 동전의 종류를 k가지라고 하면 $n-coin[k]$원에서 $coin[k]$원을 지불하여 n원을 지불하는 방법들 중 가장 적은 동전을 쓰는 방법이 됨을 알 수 있다. 따라서 다음과 같은 관계식을 만들 수 있다.

$$DT[n] \ = \begin{cases} 1 & (n = coin[k]) \\ \min(DT[n-coin[k]])+1 & \end{cases}$$

위 식에서 k는 주어진 지불 가능한 동전의 수를 의미한다. 각 동전들에 대해서 하나의 동전으로 지불 가능한 상태면 $DT[n]=1$이고, 그렇지 않으면 다시 위 식과 같이 분할할 수 있다. 이 알고리즘을 동적표를 이용하여 하향식과 상향식으로 구현한 알고리즘은 다음과 같다.

코드	참고

```
#include <cstdio>

int DT[10001], coin[10], n, m;

int min(int a, int b) { return a>b ? b:a;}

int f(int c)
{
  for(int i=0; i<n; i++)
    if(coin[i]==c) return 1;
  if(DT[c]==0)
  {
    DT[c]=987654321;
    for(int i=0; i<n; i++)
      if(c-coin[i]>0)
        DT[c]=min(DT[c], f(c-coin[i])+1);
    return DT[c];
  }
}

int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d", coin+i);
  printf("%d\n", f(m));
  return 0;
}
```

다음은 반복문을 이용하여 상향식으로 구현한 것이다.

```
#include <cstdio>

int DT[10001], coin[10], n, m;

int min(int a, int b) { return a>b ? b:a;}

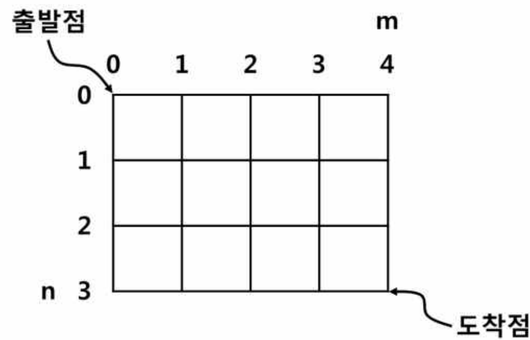
int main()
{
  scanf("%d %d", &m, &n);
  for(int i=0; i<n; i++)
    scanf("%d",coin+i), DT[coin[i]]=1;
  for(int i=coin[0]; i<=m; i++)
  {
    if(DT[i]==0)
    {
      DT[i]=987654321;
      for(int j=0; j<n; j++)
        if(i-coin[j]>0)
          DT[i] = min(DT[i], DT[i-coin[j]]+1);
    }
  }
  printf("%d\n", DT[m]);
  return 0;
}
```

이와 같이 반복문을 이용하여 상향식으로 작성하면 매우 효율적으로 동작한다. 물론 이 문제도 더 효율적인 방법으로 해결할 수 있으므로, 조금 더 좋은 아이디어에 대해서도 항상 생각할 수 있도록 하자.

문제 13

격자길(L)

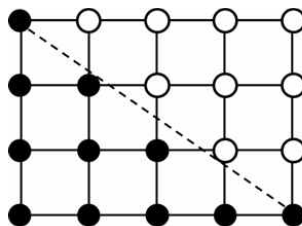
n*m 격자에서 왼쪽 위(0,0)에서 오른쪽 아래(n,m)까지 갈 수 있는 길의 수를 헤아리고자 한다.



길을 갈 때 몇 가지 제약사항이 있다.

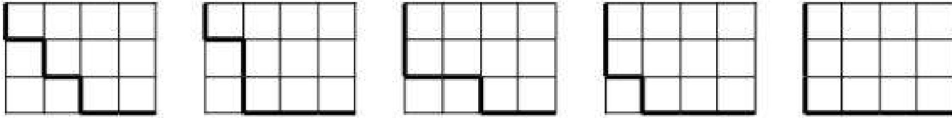
(1) 격자 위의 선을 따라간다.
(2) 아래쪽 또는 오른쪽으로만 갈 수 있다.
(3) (0,0)과 (n,m)을 잇는 대각선보다 위쪽에 있는 점들은 통과할 수 없다.
(대각선에 위치하는 점은 통과할 수 있다.)

아래의 그림에서 흰점은 통과할 수 없는 점이고 검은 점은 통과할 수 있는 점이다.



격자길(L) (계속)

예를 들어, 3*4 격자에서 갈 수 있는 길은 다음과 같이 5가지가 있다.



격자의 크기가 입력되었을 때 (0,0)부터 (n,m)까지 갈 수 있는 길의 수를 출력하는 프로그램을 작성하시오.

입력

1. 두 개의 정수 n과 m이 입력된다.
2. n은 격자의 세로 크기를, m은 격자의 가로 크기를 각각 나타낸다.

[입력값의 정의역]
1 <= n, m <= 100

출력

(0,0)에서 (n,m)까지 갈 수 있는 길의 수를 출력한다.

입력 예	출력 예
3 4	5

풀이

이 문제는 중급편에서 전체탐색으로 해결했었던 문제이다. 전체탐색으로 문제를 해결할 때의 기본 원리는 다음과 같다.

① 현재상태 - 오른쪽으로 이동
② 현재상태 - 아래쪽으로 이동
①, ②의 이동 시 주어진 영역 외부로는 나갈 수 없다.
①의 이동시 출발점과 도착점을 이은 선분의 오른쪽 위 부분(흰점이 있는 부분)으로는 이동할 수 없다.

위 원리를 지켜가면서 전체탐색을 진행할 수 있다. 이 경우 계산량이 매우 크기 때문에 맵의 크기가 커지면 해결할 수 없는 문제가 된다. 먼저 전체탐색법의 소스코드를 동적표에 적용하기 위해서는 값을 반환하는 함수로 바꾸어야한다. 이를 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <cstdio>

int n, m;

bool Available(int a, int b)
{
  return (!b || (double)n/m<=(double)a/b);
}

int f(int a, int b)
{
  if(a>n || b>m || !Available(a,b)) return 0;
  if(a==n && b==m) return 1;
  return f(a+1,b)+f(a,b+1);
}

int main()
{
  scanf("%d %d", &n, &m);
  printf("%d\n", f(0,0));
  return 0;
}
```

이 알고리즘은 중복호출이 많이 발생되므로 실행시간이 너무 오래 걸린다. 하지만 탐색을 담당하는 함수 f가 값을 반환하는 함수이므로 동적표 적용이 가능하다. 동적표 DT를 다음과 같이 정의하여 적용해보자.

$$DT[a][b] \ = \ f(a,\ b)\text{의 값을 기록}$$

이를 이용하여 구현한 소스코드는 다음과 같다.

```
#include <cstdio>

int n, m, DT[110][110];

bool Available(int a, int b)
{
  return (!b || (double)n/m<=(double)a/b);
}

int f(int a, int b)
{
  if(DT[a][b]==0)
  {
    if(a>n || b>m || !Available(a,b)) DT[a][b]=0;
    else if(a==n && b==m) DT[a][b]=1;
    else DT[a][b]=f(a+1,b)+f(a,b+1);
  }
  return DT[a][b];
}

int main()
{
  scanf("%d %d", &n, &m);
  printf("%d\n", f(0,0));
  return 0;
}
```

이와 같이 구현할 경우 효율이 매우 좋아진다. 계산량은 대략적으로 nm정도로 충분히 제한 시간 이내에 해결할 수 있는 알고리즘이 된다.

다음으로 동적계획법으로 이 문제를 해결해보자. 위 방법으로 한 번 해결했기 때문에, 어렵지 않게 관계식을 만들 수 있다. 먼저 $DT[a][b]$를 다음과 같이 정의하자.

$DT[a][b]$ = (0, 0)으로부터 (a, b)까지 이동가능한 경우의 수

이 정의를 이용하여 $DT[n][m]$을 구하면 된다. 먼저 관계식을 설계하기 위하여 (a, b)로 이동할 수 있는 칸은 (a, $b-1$) 또는 ($a-1$, b)뿐이다. 하지만 이미 (a, $b-1$) 또는 ($a-1$, b)까지의 경우의 수를 알고 있다고 가정하면 다음과 같은 관계식을 만들 수 있다.

$$DT[a][b] = DT[a-1][b] + DT[a][b-1]$$

그리고 (a, b)칸의 점이 흰색일 경우에는 그 칸은 이동 불가하므로 $DT[a][b]=0$이 된다. 따라서 다음과 같은 관계식을 만들 수 있다.

$$DT[a][b] = \begin{cases} 1 & (a=0,\ b=0) \\ 0 & (\frac{a}{b} < \frac{n}{m}) \\ DT[a-1][b] + DT[a][b-1] & \end{cases}$$

이 식을 이용하여 하향식 재귀함수와 메모이제이션을 이용한 소스코드는 다음과 같다.

```
#include <cstdio>

int n, m, DT[110][110];

bool Available(int a, int b)
{
  return (!b || (double)n/m<=(double)a/b);
}

int f(int a, int b)
{
  if(DT[a][b]==0)
  {
    if(a==0 && b==0) DT[a][b]=1;
    else if(!Available(a,b)) DT[a][b]=0;
    else DT[a][b]=f(a-1,b)+f(a,b-1);
  }
  return DT[a][b];
}

int main()
{
    scanf("%d %d", &n, &m);
    printf("%d\n", f(n,m));
    return 0;
}
```

위 식을 상향식으로 구현한 소스코드는 다음과 같다.

```
#include <cstdio>

int n, m, DT[110][110];

bool Available(int a, int b)
{
  return (!b || (double)n/m<=(double)a/b);
}

int main()
{
  scanf("%d %d", &n, &m);
  for(int i=0; i<=n; i++)
    for(int j=0; j<=m; j++)
      if(i==0 && j==0) DT[i][j]=1;
      else if(!Available(i, j)) DT[i][j]=0;
      else DT[i][j]=DT[i-1][j]+DT[i][j-1];
  printf("%d\n", DT[n][m]);
  return 0;
}
```

위 2가지 방법의 전체적인 계산량은 같지만 반복문을 이용한 동적계획법이 함수의 호출이 없기 때문에 더 효율적이므로 가능하면 동적계획법으로 코딩하는 연습을 하는 것이 좋다.

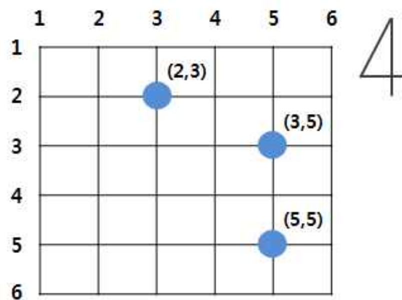
문제 14

경찰차(L)

어떤 도시의 중심가는 n개의 동서방향 도로와 n개의 남북방향 도로로 구성되어 있다.

모든 도로에는 도로 번호가 있으며 남북방향 도로는 왼쪽부터 1에서 시작하여 n까지 번호가 할당되어 있고 동서방향 도로는 위부터 1에서 시작하여 n까지 번호가 할당되어 있다. 또한 동서방향 도로 사이의 거리와 남북방향 도로 사이의 거리는 모두 1이다.

동서방향 도로와 남북방향 도로가 교차하는 교차로의 위치는 두 도로의 번호의 쌍인 (동서방향 도로 번호, 남북방향 도로 번호)로 나타낸다. n이 6인 경우의 예를 들면 다음과 같다.



이 도시에는 두 대의 경찰차가 있으며 두 차를 경찰차1과 경찰차2로 부른다. 처음에는 항상 경찰차1 은 (1, 1)의 위치에 있고 경찰차2는 (n, n)의 위치에 있다.

경찰 본부에서는 처리할 사건이 있으면 그 사건이 발생된 위치를 두 대의 경찰차 중 하나에 알려 주고, 연락 받은 경찰차는 그 위치로 가장 빠른 길을 통해 이동하여 사건을 처리한다(단, 하나의 사건은 한 대의 경찰차가 처리한다.).

그리고 사건을 처리한 경찰차는 경찰 본부로부터 다음 연락이 올 때까지 처리한 사건이 발생한 위치에서 기다린다. 경찰 본부에서는 사건이 발생한 순서대로 두 대의 경찰차에 맡기려고 한다.

경찰차(L) (계속)

처리해야 될 사건들은 항상 교차로에서 발생하며 경찰 본부에서는 이러한 사건들을 나누어 두 대의 경찰차에 맡기되, 두 대의 경찰차들이 이동하는 거리의 합을 최소화하도록 사건을 맡기려고 한다.

예를 들어 앞의 그림처럼 n=6인 경우, 처리해야 하는 사건들이 3개 있고 그 사건들이 발생된 위치를 순서대로 (3, 5), (5, 5), (2, 3)이라고 하자.

(3, 5)의 사건을 경찰차2에 맡기고 (5, 5)의 사건도 경찰차2에 맡기며, (2, 3)의 사건을 경찰차1에 맡기면 두 차가 이동한 거리의 합은 4 + 2 + 3 = 9가 되고, 더 이상 줄일 수는 없다.

처리해야 할 사건들이 순서대로 주어질 때, 두 대의 경찰차가 이동하는 거리의 합을 최소화하는 프로그램을 작성하시오.

입력

입력 파일의 첫째 줄에는 동서방향 도로의 개수를 나타내는 정수 n(3≤n≤1,000)이 주어진다.

둘째 줄에는 처리해야 하는 사건의 개수를 나타내는 정수 m(1≤m≤15)가 주어진다.

셋째 줄부터 (m+2)번째 줄까지 사건이 발생된 위치가 한 줄에 하나씩 주어진다. 경찰차들은 이 사건들을 주어진 순서대로 처리해야 한다.

각 위치는 동서방향 도로 번호를 나타내는 정수와 남북방향 도로 번호를 나타내는 정수로 주어지며 두 정수 사이에는 빈 칸이 하나 있다. 두 사건이 발생한 위치가 같을 수 있다.

출력

첫째 줄에 두 경찰차가 이동한 총 거리를 출력한다.

입력 예	출력 예
6 3 3 5 5 5 2 3	9

출처: 한국정보올림피아드(2003 전국본선 중등부)

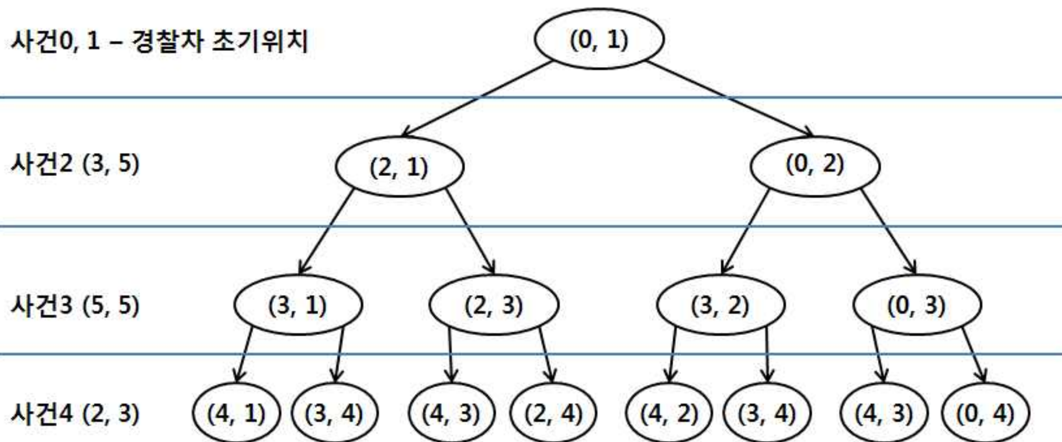
풀이

이 문제는 한국정보올림피아드 전국대회 중등부에 출제되었던 문제로 당시 중등부 수준으로 볼 때 난이도가 제법 높았던 문제이다. 하지만 중급편에서 설계했던 탐색구조를 잘 이용하면 크게 어렵지 않게 해결할 수 있는 문제이므로, 잘 익힐 수 있도록 하자.

중급편에서 이 문제의 탐색구조를 다음과 같이 정의했다.

> $solve(a,\ b,\ d)\ =$
> "$\max(a,\ b)$번 사건까지 처리하면서 d만큼 이동한 후, 1번 경찰차는 a사건의 위치에, 2번 경찰차는 b사건의 위치에 있는 상태"

그리고 다음과 같이 탐색을 진행하여 해를 구했다.



이 풀이에서 탐색함수 $solve$를 정수를 반환하는 함수인 f로 정의를 바꾸면 동적표를 적용하여 중복호출을 배제할 수 있으므로, 계산량을 획기적으로 개선할 수 있다. 먼저 탐색함수를 정수를 반환하는 함수로 바꾼 소스코드는 다음과 같다.

```c
#include <stdio.h>

int E[1010][2], n, m, ans = 987654321;

int min(int a, int b){ return a>b?b:a; }
int abs(int a){ return a>0?a:-a; }

int dis(int a, int b)
{
  return abs(E[a][0]-E[b][0])+abs(E[a][1]-E[b][1]);
}

int f(int a, int b)
{
  int next=(a>b?a:b)+1;
  if(next>=m+2) return 0;
  return min(f(next,b)+dis(a,next),f(a,next)+dis(b,next));
}

int main()
{
  scanf("%d %d",&n,&m);
  E[0][0]=E[0][1]=1;
  E[1][0]=E[1][1]=n;
  for(int i=2; i<m+2; i++)
    scanf("%d %d", &E[i][0], &E[i][1]);
  printf("%d\n", f(0,1));
  return 0;
}
```

위 알고리즘은 탐색함수가 값을 정수형으로 반환하므로, 이를 바로 동적표에 저장할 수 있는 형태이다. 따라서 동적표 DT를 다음과 같이 정의할 수 있다.

$$DT[a][b] \ = \ f(a,\ b)\text{를 저장한 값}$$

이 방법으로 메모이제이션 기법을 진행하면 모든 함수 f에 대한 중복호출을 배제할 수 있으므로 시간을 줄일 수 있다.

동적표를 적용한 소스코드는 다음과 같다.

코드 | 참고

```
#include <stdio.h>

int E[1010][2], n, m, ans=987654321, DT[1100][1100];

int min(int a, int b){ return a>b ? b:a;}

int abs(int a){ return a>0 ? a:-a;}

int dis(int a, int b)
{
  return abs(E[a][0]-E[b][0])+abs(E[a][1]-E[b][1]);
}

int f(int a, int b)
{
  if(DT[a][b]==0)
  {
    int next=(a>b ? a:b)+1;
    if(next>=m+2) DT[a][b]=0;
    else
DT[a][b]=min(f(next,b)+dis(a,next),f(a,next)+dis(b,next));
  }
  return DT[a][b];
}

int main()
{
  scanf("%d %d", &n, &m);
  E[0][0]=E[0][1]=1;
  E[1][0]=E[1][1]=n;
  for(int i=2; i<m+2; i++)
    scanf("%d%d", &E[i][0], &E[i][1]);
  printf("%d\n", f(0,1));
  return 0;
}
```

이와 같이 동적표를 적용하면 어렵지 않게 해결할 수 있는 문제가 된다.

다음으로 이 문제를 상향식 동적계획법으로 해결하기 위하여 관계식을 설계해보자. 먼저 동적테이블 DT를 다음과 같이 정의하자.

$DT[a][b]$ = 1번 경찰차는 a사건, 2번 경찰차는 b사건을 해결한 최소 이동거리

이 정의를 만족시키기 위하여 관계를 만들려면 몇 가지 사항을 고려해야 한다. 먼저 a, b중 a의 값이 클 경우는 이번에 해결한 사건이 a사건이라는 의미이고, b의 값이 클 경우는 b사건을 해결한 경우이다. 왜냐하면 사건은 오름차순으로 처리되어야하기 때문이다.

따라서 먼저 위 식의 관계를 찾기 위해서는 $a > b$인 경우와 $b < a$인 경우를 나누어 처리해야 한다. 물론 두 경찰차가 같은 사건을 처리하고 같은 위치에 있을 수 없기 때문에 $a = b$인 경우는 없다.

먼저 $a > b$인 경우에 대해서 살펴보자. 이 경우는 a번 사건을 해결한 경우이므로 반드시 1번 경찰차가 이동해서 사건을 처리한 경우이다. 따라서 다음과 같은 관계식을 만들 수 있다.

$$DT[a][b] = \begin{cases} \min(DT[k][b] + dist(k,\ a)) & (0 \le k < b,\ a > b) \\ DT[a-1][b] + dist(a-1,\ a) & (b < a-1) \end{cases}$$

다음으로 $a < b$인 경우는 반드시 2번 경찰차가 b번 사건의 위치로 이동하는 경우이므로 다음과 같은 관계식을 만들 수 있다.

$$DT[a][b] = \begin{cases} \min(DT[a][k] + dist(k,\ b)) & (0 < k < a,\ a < b) \\ DT[a][b-1] + dist(b-1,\ b) & (a < b-1) \end{cases}$$

위 두 개의 식에서 k의 범위에 차이가 나는 이유는 1번 경찰차는 사건 0으로부터 시작하고, 2번 경찰차는 사건 1로부터 시작하기 때문이다. 이 식을 정리하면 다음과 같다.

$$DT[a][b] = \begin{cases} 0 & (a = 0,\ b = 1) \\ \min(DT[k][b] + dist(k,\ a)) & (0 \le k < a,\ a > b) \\ DT[a-1][b] + dist(a-1,\ a) & (b < a-1) \\ \min(DT[a][k] + dist(k,\ b)) & (0 < k < b,\ b > a) \\ DT[a][b-1] + dist(b-1,\ b) & (a < b-1) \end{cases}$$

이 식을 이용하여 구현한 소스코드는 다음과 같다.

```
#include <cstdio>
#include <cstring>
#define INF 0x3f3f3f
int E[1010][2], n, m, ans=INF, DT[1100][1100];

int min(int a, int b){ return a>b ? b:a;}
int abs(int a){ return a>0 ? a:-a;}
int dist(int a, int b)
{
  return abs(E[a][0]-E[b][0])+abs(E[a][1]-E[b][1]);
}
int main()
{
  scanf("%d %d", &n, &m);
  E[0][0]=E[0][1]=1, E[1][0]=E[1][1]=n;
  for(int i=2; i<m+2; i++)
    scanf("%d %d", &E[i][0], &E[i][1]);
  memset(DT, 0x3f, sizeof(DT)), DT[0][1]=0;
  for(int i=0; i<m+2; i++)
    for(int j=1; j<m+2; j++)
    {
      if(i==j) DT[i][j]==INF;
      else if(i>j)
      {
        if(i-1>j) DT[i][j]=DT[i-1][j]+dist(i-1,i);
        else for(int k=0; k<j; k++)
          DT[i][j]=min(DT[i][j],DT[k][j]+dist(k,i));
      }
      else
      {
        if(j-1>i) DT[i][j]= DT[i][j-1]+dist(j-1,j);
        else for(int k=1; k<i; k++)
          DT[i][j]=min(DT[i][j],DT[i][k]+dist(k,j));
      }
    }
  for(int i=0; i<m+2; i++)
    ans = min(ans, min(DT[i][m+1], DT[m+1][i]));
  printf("%d\n", ans);
  return 0
}
```

물론 DT를 다른 형태로 정의해서 해결하는 방법도 있으므로 다양한 방법으로 연구해 볼 수 있기 바란다. 그리고 이 문제의 특징은 상향식으로 갈 때, 계산량이 다소 증가하는 경향이 있으므로 항상 상향식이 좋은 것만은 아니다. 물론 k값을 미리 계산하여 처리하는 방법을 이용하면 효율을 높일 수도 있다.

문제 15

돌다리 건너기(L)

절대반지를 얻기 위하여 반지원정대가 출발한다. 원정대가 지나가야할 다리는 두 개의 인접한 돌다리로 구성되어 있다. 하나는 <악마의 돌다리>이고 다른 하나는 <천사의 돌다리>이다. 아래 그림 1은 길이가 6인 다리의 한 가지 모습을 보여준다.

그림에서 위의 가로줄은 <악마의 돌다리>를 표시하는 것이고 아래의 가로줄은 <천사의 돌다리>를 표시한다. 두 돌다리의 길이는 항상 동일하며, 각 칸의 문자는 해당 돌에 새겨진 문자를 나타낸다. 두 다리에 새겨진 각 문자는 {R, I, N, G, S} 중 하나이다.

출발	R	I	N	G	S	R	도착
	G	R	G	G	N	S	

반지원정대가 소유하고 있는 마법의 두루마리에는 <악마의 돌다리>와 <천사의 돌다리>를 건너갈 때 반드시 순서대로 밟고 지나가야할 문자들이 적혀있다. 이 순서대로 지나가지 않으면 돌다리는 무너져, 반지원정대는 화산 속으로 떨어지게 된다. 다리를 건널 때 다음의 제한조건을 모두 만족하면서 건너야 한다.

(1) 왼쪽(출발지역)에서 오른쪽(도착지역)으로 다리를 지나가야 하며, 반드시 마법의 두루마리에 적힌 문자열의 순서대로 모두 밟고 지나가야 한다.
(2) 반드시 <악마의 돌다리>와 <천사의 돌다리>를 번갈아가면서 돌을 밟아야 한다. 단, 출발은 어떤 돌다리에서 시작해도 된다.
(3) 반드시 한 칸 이상 오른쪽으로 전진해야하며, 건너뛰는 칸의 수에는 상관이 없다.

만일 돌다리의 모양이 그림 1과 같고 두루마리의 문자열이 "RGS"라면 돌다리를 건너갈 수 있는 경우는 다음의 3가지뿐이다. (아래 그림에서 큰 문자는 밟고 지나가는 돌다리를 나타낸다.)

출발	**R**	I	N	G	**S**	R	도착
	G	R	**G**	G	N	S	

출발	**R**	I	N	G	**S**	R	도착
	G	R	G	**G**	N	S	

출발	R	I	N	**G**	S	R	도착
	G	**R**	G	G	N	**S**	

돌다리 건너기(L) (계속)

아래의 세 방법은 실패한 방법이다.

출발	**R**	I	N	G	S	R	도착
	G	R	**G**	G	N	S	

출발	**R**	I	N	**G**	S	R	도착
	G	R	G	G	N	**S**	

출발	**R**	I	N	G	**S**	R	도착
	G	R	G	G	N	S	

왜냐하면 첫 번째는 문자열 "RGS"를 모두 밟고 지나가야 하는 조건 (1)을 만족하지 않으며, 두 번째는 번갈아가면서 돌을 밟아야 하는 조건 (2)를, 세 번째는 앞으로 전진을 하여야하는 조건 (3)을 만족하지 않기 때문이다.

마법의 두루마리에 적힌 문자열과 두 다리의 돌에 새겨진 문자열이 주어졌을 때, 돌다리를 통과할 수 있는 모든 가능한 방법의 수를 계산하는 프로그램을 작성하시오. 예를 들어, 그림 1의 경우는 통과하는 방법이 3가지가 있으므로 3을 출력해야 한다.

입력

첫째 줄에는 마법의 두루마리에 적힌 문자열(R, I, N, G, S로만 구성된)이 주어진다. 이 문자열의 길이는 최소 2, 최대 10이다. 그 다음 두 줄에는 각각 <악마의 돌다리>와 <천사의 돌다리>를 나타내는 같은 길이의 문자열이 주어진다. 그 길이는 5 이상, 20 이하이다.

출력

출력 파일에 마법의 두루마리에 적힌 문자열의 순서대로 다리를 건너갈 수 있는 방법의 수를 출력한다. 그러한 방법이 없으면 0을 출력한다. 모든 테스트 데이터에 대한 출력결과는 2^31 - 1 이하이다.

입력 예	출력 예
RGS RINGSR GRGGNS	3
RINGS SGNIRSGNIR GNIRSGNIRS	0
GG GGGGRRRR IIIIGGGG	16

출처: 한국정보올림피아드(2004 전국본선 고등부)

풀이

돌다리를 밟는 순서와 돌다리의 상태가 주어질 때, 돌다리를 건널 수 있는 모든 경우의 수를 찾는 문제이다. 만족해야 하는 조건은 다음과 같다.

- 두루마리에 적힌 순서대로 지나가야 한다.
- 악마의 돌다리와 천사의 돌다리를 번갈아 밟고 지나가야 한다.
- 출발은 어느 돌다리에서 하던지 상관없다.
- 반드시 오른쪽으로 한 칸 이상씩 전진해야 한다.
- 건너뛰는 칸의 수는 제한이 없다.

두루마리	R	G	S

악마의 돌다리	R	I	N	G	S
천사의 돌다리	G	R	G	G	N

만약에 악마의 돌다리에서 첫 번째 R을 선택하면, 반대편 천사의 돌다리에서 두 번째 G를 선택해야 한다. 이때 천사의 돌다리를 보면 G가 하나가 아님을 알 수 있다.

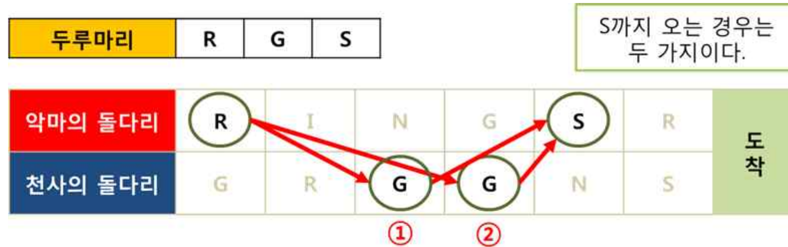
결국 ①번과 ②번의 G도 가능하기 때문에 어느 돌다리가 하나 선택되었다면 다음 돌다리로 갈 때 가능한 모든 경우에 대해서 계산하여야 한다.

그리고 최종적으로 두루마리에 남아 있는 문자가 없는 경우는 가능한 경우를 찾은 것이고 두루마리에 문자가 남았는데 돌다리를 모두 지났다면 불가능한 경우이다.

전체 탐색에서 문제를 자세히 보면 상당히 많은 중복을 볼 수가 있다. 예를 들어 아래의 그림과 같이 악마의 돌다리 R에서 시작해서 천사의 돌다리 ①번으로 가고 다시 악마의 돌다리 S로 오는 경우와 천사의 돌다리 ②번으로 가서 다시 악마의 돌다리 S로 오는 경우가 있다.

이때 ①번에서 S로 간 경우 S를 시작으로 그 이하의 경우를 구하게 된다. 그런데 ②번에서 S로 간 경우 역시 또 S를 시작으로 경우의 수를 구하기 때문에 중복이 발생하게 된다. 이를 배제하면 시간 안에 문제를 해결할 수 있다. 이때 S를 시작하여 구한 경우의 수를 저

장해 두는 메모이제이션 기법을 활용하면 된다.



위의 아이디어를 이용한 소스코드는 다음과 같다.

```
#include <stdio.h>

char rol[30], dol[2][120];
int DT[2][120][30];

int f(int dolnum, int dolpos, int rolpos)
{
  if(DT[dolnum][dolpos][rolpos]==0)
  {
    if(rol[rolpos]=='\0')
      DT[olnum][dolpos][rolpos]=1;
    for(int i=dolpos; dol[1-dolnum][i]!='\0'; i++)
      if(dol[1-dolnum][i]==rol[rolpos])
        DT[olnum][dolpos][rolpos]+=f(1-dolnum,i+1,rolpos+1);
  }
  return DT[dolnum][dolpos][rolpos];
}

int main()
{
  scanf("%s %s %s", rol, dol[0], dol[1]);
  printf("%d", f(0,0,0)+f(1,0,0));
  return 0;
}
```

기본적으로 중급에서 소개했던 알고리즘에서 메모이제이션 기법을 사용하기 위한 배열로 DT를 만들고 돌다리 종류, 돌다리의 방문위치, 두루마리 방문위치의 정보에 맞게 선언한다.

4행에서 각 조건에 해당하는 값이 구해져 있으면 해당 배열의 값을 반환한다.

12행~14행에서 돌다리종류가 번갈아 가면서 선택을 위해 코드를 1-dolnum으로 변경하여 코드를 단순화 시켰다. 예를 들어 dolnum가 0일 경우 1-0이므로 1이 되고, 1이면 1-1이므로 0으로 된다. 따라서 1은 0으로 0은 1로 돌다리 종류가 변경된다.

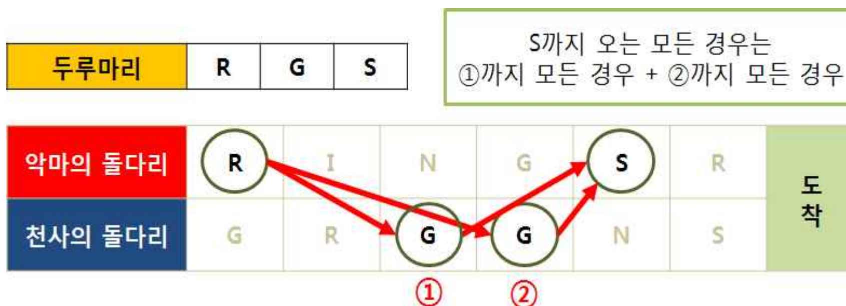
16행에서 단순히 합한 결과를 반환하기 전에 배열의 테이블에 저장하여 반환한다.

따라서 이 알고리즘의 계산량은 배열의 크기인 $dolnum \times dolpos \times rolpos$의 크기에 f함수의 재귀호출에 따른 실행시간(+a)이다. 따라서 계산량은 배열의 크기와 같다.

이번에는 새로운 방법으로 알고리즘을 만들어보자. 위의 아이디어를 바탕으로 테이블로 각각의 경우를 계산하면 빠르게 완성할 수 있다. 다음과 같이 테이블을 정의하면 다음과 같다.

$DT[s][r]$ = s돌다리에서 r번 문자로 끝나는 모든 경우

결국 최종 구하고자 하는 문자로 끝나는 경우인 f(r)=DT[1][r]+DT[0][r]이 된다. 이 때, DT[0][r]는 가능한 모든 DT[1][r−1]의 합이 되고 DTs[1][r]는 가능한 모든 DTs[0][r−1]의 합이 되게 된다.



이 아이디어를 이용한 소스코드는 다음과 같다.

줄	코드	참고

```c
#include <stdio.h>
#include <string.h>

char rol[30], dol[2][120];
int DT[2][30], rc;

int f(int k)
{
  for(int i=0; dol[1][i]; i++)
  {
    for(int t=k-1; t>=0; t--)
    {
      if(dol[0][i]==rol[t]) DT[1][t+1]+=DT[0][t];
      if(dol[1][i]==rol[t]) DT[0][t+1]+=DT[1][t];
    }
  }
  return DT[0][k]+DT[1][k];
}

int main()
{
  scanf("%s %s %s", rol, dol[0], dol[1]);
  rc=strlen(rol);
  DT[0][0]=1;
  DT[1][0]=1;
  printf("%d", f(rc));
  return 0;
}
```

문제 16

선물(L)

길동이는 세쌍둥이의 첫째이다. 길순이가 둘째이고, 길삼이가 막내이다. 길동 3남매의 생일을 맞이하여 전국 각지에서 친지들이 보내온 수많은 선물이 도착하였다.

길동이 부모는 이 선물들을 길동이 3남매에게 어떻게 나누어 줄 것인가로 고민하고 있다. 선물의 크고 작음 때문에 발생될 수도 있는 남매간의 다툼을 미연에 방지하고자 길동이 가족은 다음과 같이 나누기로 결정하였다.

(1) 선물의 내용을 미리 보지 않고 부피만을 기준으로 배분한다.
(2) 한 사람이 가지는 선물의 개수는 배분의 기준이 아니다.
(3) 선물이 공평하게 나누어질 수 있도록 3남매가 가지는 선물들의 부피의 합계 차이가 최소가 되도록 한다.
(4) 선물의 부피가 똑같이 나누어지지 못하는 경우에는 길동-길순-길삼의 순으로 합계 부피가 많도록 배분한다.
(5) 3남매가 가지게 되는 부피가 결정되면, 길삼-길순-길동의 순으로 선물을 선택한다.

우리가 길동 부모의 수고를 덜어주고자 길동이 3남매가 가지게 될 선물의 부피를 계산하고자 한다. 선물 부피에 따른 선물 배분의 세부적인 조건은 다음과 같다.

조건 1: 아래의 d가 최소가 되도록 한다.

d = (길동 선물의 부피 합) − (길삼 선물의 부피 합)

조건 2: 같은 d가 되는 배분 방법이 여럿 존재하는 경우에는 길동의 선물의 부피 합이 적은 방법을 선택한다.

선물(L) (계속)

예를 들어, 선물이 6개이고 그 부피가 다음과 같다면,

6, 4, 4, 4, 6, 9

길동은 부피의 합계가 12, 길순은 12, 길삼은 9를 가지도록 배분하면 조건 1에 따라 12-9=3로 최소가 된다.

(길동 13, 길순 10, 길삼 10으로 배분하는 방법도 13-10=3으로 차이가 3이 되지만, 조건 2에 따라 답이 되지 못한다.)

선물의 부피가 입력되었을 때 3남매에게 나누어줄 선물의 합계 부피를 구하는 프로그램을 작성하시오.

입력

1. 첫 줄에 선물의 개수를 나타내는 정수 n이 입력된다. (3≤n≤20)
2. 다음 줄에 선물의 부피를 나타내는 n개의 정수가 공백으로 분리되어 입력된다.
3. 선물의 부피는 0보다 크고 100보다 작다

출력

1. 길동 3남매가 가지게 될 선물의 합계 부피를 출력한다.
2. 길동, 길순, 길삼의 순으로 3개의 정수를 하나의 공백으로 분리하여 출력한다.

입력 예	출력 예
6 6 4 4 4 6 9	12 12 9
3 2 10 1	10 2 1
9 1 1 1 4 6 1 1 1 1	6 6 5

풀이

중급편에서 $k=15$인 문제를 다루어보았다. 중급편에서 다룬 문제는 전체탐색법으로 해결할 수 있는 크기였으나, 이번 문제는 $k=20$이므로 전체탐색법으로 해결하기는 쉽지 않은 크기이다.

따라서 새로운 방법으로 접근해야 한다. 이 문제는 동적계획법으로 해결할 수 있는 문제이다. 하지만, 지금까지 설계한 동적계획법과는 접근법이 약간 다른 문제이다. 이 문제에서 각 선물의 부피가 100이고 선물의 개수가 20개이므로 모든 선물 부피의 합이 2000을 초과하지 않는다.

따라서 2000이라는 값을 이용하여 동적계획법으로 접근할 수 있는 아이디어를 생각할 수 있다.

일단 전체 선물의 부피를 W라고 하자. 그리고 길동이 받는 총 선물의 부피를 a, 길순이가 받는 총 선물의 부피를 b, 길삼이가 받는 총 선물의 부피를 c라고 하면 다음 관계가 성립된다.

$$W = a + b + c$$

그리고 W를 알고 있으므로 a값은 기억하지 않더라도 다음 식을 이용하여 a값을 구할 수 있다.

$$a = W - (b + c)$$

자 이제 관계식을 만들기 위해 DT를 다음과 같이 정의하자.

$DT[k][b][c]$ =현재 k개의 선물을 받았을 때, 길순이가 부피 b, 길삼이가 부피 c를 받을 수 있는 상태가 가능하면 1, 그렇지 않으면 0

이번에 정의한 동적표는 약간 특이한 면을 가지고 있다. 이는 탐색구조와 비슷한데, 탐색구조에서 가능한 상태의 수가 $2000 \times 2000 \times 2000$이고 그 중 한 명은 기억하지 않아도 전체 부피를 통해서 구할 수 있으므로 실제 2000×2000의 상태로 삼형제의 선물의 부피 상태를 동적표로 나타낼 수 있다.

그리고 현재까지 몇 개의 선물을 분배했는지도 필요하므로, $20\times2000\times2000$의 테이블이면 모든 상태를 나타낼 수 있고, 그 상태가 가능한지 여부를 저장하여 계산량을 줄일 수 있다.

다음으로 관계식을 유도해보자. 만약 k번째 선물을 받은 후의 모든 상태는 $k-1$번째 선물을 받은 상태로부터 구할 수 있다.

k번째 선물의 무게를 w_k라고 하면 이 선물을 길동이가 받을 경우는 길순이와 길삼이의 부피는 변화가 없으므로 $DT[k][a][b] = DT[k-1][a][b]$와 같은 식으로 표현할 수 있다. 그리고 길순이와 길삼이도 선물을 받아서 $DT[k][a][b]$의 상태를 만들 수 있으므로 세 명 중 한 명이라도 이 값이 1이면 $DT[k][a][b]$의 값은 1이 된다. 이를 모든 a와 b에 대해서 처리하면 동적표를 채울 수 있다.

따라서 다음과 같은 관계식을 유도할 수 있다.

$$DT[k][a][b] = \begin{cases} 1 & (k=0,\ a=0,\ b=0) \\ DT[k-1][a][b] \text{ or } DT[k-1][a-w_k][b] \text{ or } DT[k-1][a][b-w_k] & (a-w_k \ge 0,\ b-w_k \ge 0) \end{cases}$$

그리고 이 동적표를 이용하여 해를 구할 수 있다. 해는 $DT[n][a][b]$의 모든 a와 b를 통하여 구할 수 있다.

이 식을 이용하여 구현한 소스코드는 다음과 같다.

```
#include <cstdio>

int n, W, G[21], A, B, C, ans=987654321;
bool DT[21][2001][2001];
int main()
{
  scanf("%d", &n);
  for(int i=1; i<=n; i++)
    scanf("%d", G+i), W+=G[i];
  DT[0][0][0]=true;
  for(int i=1; i<=n; i++)
    for(int a=0; a<=2000; a++)
      for(int b=0; b<=2000; b++)
        DT[i][a][b]=(DT[i-1][a][b] ||
        (a-G[i]<0?false:DT[i-1][a-G[i]][b]) ||
        (b-G[i]<0?false:DT[i-1][a][b-G[i]]));
  for(int a=0; a<=2000; a++)
    for(int b=0; b<=2000; b++)
      if(DT[n][a][b])
      {
        if(W-(a+b)>=a && a>=b && W-(a+b)-b<=ans)
          ans=W-(a+b)-b, A=W-(a+b), B=a, C=b;
      }
  printf("%d %d %d\n", A, B, C);
  return 0;
}

```

위 알고리즘은 주어진 모든 입력에 대해서 정확한 해를 출력한다. 단, 시간이 생각보다 오래 걸린다는 단점이 있다.

위 알고리즘에서 필요 없는 연산을 줄여서 효율을 높여보자. 위 알고리즘은 길순이와 길삼이의 선물의 부피를 동적표에 저장하고 있다. 길동이가 받은 선물의 부피를 a, 길순이가 받은 선물의 부피를 b, 길삼이가 받은 선물의 부피를 c라 할 때, a, b, c 사이에는 다음과 같은 부등식이 성립한다.

$$a \le b \le c$$

전체 선물의 최대 부피가 2000이므로 $\frac{2000}{3} = 666.666\cdots \simeq 667$ 이므로 길순이와 길삼이의 선물의 부피는 절대로 667을 넘을 수 없음이 명백하다. 따라서 동적테이블의 크기를 $20 \times 2000 \times 2000$으로 설정하지 말고, $20 \times 667 \times 667$로 설정해도 문제를 해결하는 데는 지장이 없다. 따라서 다음과 같이 수정하면 효율이 10배 이상 향상된다.

코드	참고

```
#include <cstdio>

int n, W, G[21], A, B, C, ans=987654321;
bool DT[21][668][668];
int main()
{
  scanf("%d", &n);
  for(int i=1; i<=n; i++)
    scanf("%d", G+i), W+=G[i];

  DT[0][0][0]=true;
  for(int i=1; i<=n; i++)
    for(int a=0; a<=667; a++)
      for(int b=0; b<=667; b++)
        DT[i][a][b]=(DT[i-1][a][b] ||
        (a-G[i]<0?false:DT[i-1][a-G[i]][b]) ||
        (b-G[i]<0?false:DT[i-1][a][b-G[i]]));
  for(int a=0; a<=667; a++)
    for(int b=0; b<=667; b++)
      if(DT[n][a][b])
      {
        if(W-(a+b)>=a && a>=b && W-(a+b)-b<=ans)
          ans=W-(a+b)-b, A=W-(a+b), B=a, C=b;
      }
  printf("%d %d %d\n", A, B, C);
  return 0;
}
```

다음으로 이 알고리즘의 메모리 사용량을 획기적으로 줄여보자. 이 알고리즘에서 668×668크기의 메모리를 20개 사용한다. 하지만 잘 분석해보면 결국 n번째 668×668을 채우기 위해서 $n-1$번째 표만 있으면 된다.

즉 668×668크기의 동적표를 2개만 활용하여 모든 값을 저장하고 활용할 수 있다. 이를 적용한 알고리즘은 다음과 같다.

```
#include <cstdio>

int n, W, G[21], A, B, C, ans=987654321;
bool DT[2][668][668];
int main()
{
  scanf("%d", &n);
  for(int i=1; i<=n; i++)
    scanf("%d", G+i), W+=G[i];

  DT[0][0][0]=true;
  for(int i=1; i<=n; i++)
    for(int a=0; a<=667; a++)
      for(int b=0; b<=667; b++)
        DT[i%2][a][b]=(DT[(i-1)%2][a][b] ||
        (a-G[i]<0?false:DT[(i-1)%2][a-G[i]][b]) ||
        (b-G[i]<0?false:DT[(i-1)%2][a][b-G[i]]));
  for(int a=0; a<=667; a++)
    for(int b=0; b<=667; b++)
      if(DT[n%2][a][b])
      {
        if(W-(a+b)>=a && a>=b && W-(a+b)-b<=ans)
          ans=W-(a+b)-b, A=W-(a+b), B=a, C=b;
      }
  printf("%d %d %d\n", A, B, C);
  return 0;
}
```

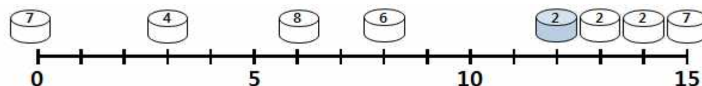
이와 같은 방법으로 처리하면 메모리를 획기적으로 줄일 수 있다. 이러한 방법을 토글링(toggling)기법이라고 한다. 동적계획법으로 알고리즘을 작성할 때에는 많은 양의 메모리를 사용하는 경우가 많다. 따라서 이러한 기법이 많은 도움이 될 수 있다.

문제 17

공주 구하기(L)

유시 섬에서 한가롭게 소풍을 즐기던 다리오와 오렌지 공주.

다리오가 잠시 자리를 비운 사이에 못된 악당 후퍼가 공주를 데리고 도망가 버렸다. 다리오는 후퍼가 오렌지 공주를 숨겨놓은 후퍼 섬으로 여행을 떠난다.



유시 섬에서 후퍼 섬까지 가기 위해서는 중간에 있는 여러 개의 섬을 거쳐 가야 한다. 유시 섬과 후퍼 섬을 포함한 모든 섬들은 유시 섬과 후퍼 섬을 지나는 직선상에 있다. 위 그림에서, 섬들을 나타내는 동그라미 아래에 있는 눈금자가 각각의 섬이 유시 섬과 몇 km나 떨어져 있는지를 나타낸다.

가장 왼쪽에 있는 섬이 유시 섬이고, 가장 멀리 있는 후퍼 섬은 15km 떨어져 있다. 한 섬에서 다른 섬으로 건너가기 위해서는 섬마다 하나씩 있는 스프링 발판을 밟아 점프해야 한다. 이 스프링 발판은 내구성이 약해서 한 번 사용하면 부서져 버린다.

이 때문에, 시작점인 유시 섬을 제외한 모든 섬들은 두 번 이상 방문하면 안 된다. 스프링 발판들의 스프링의 세기는 모두 다르다. 섬을 나타내는 동그라미에 쓰여 있는 숫자는 스프링 발판을 딛고 점프했을 때 가장 멀리 도달할 수 있는 거리를 나타낸다.

가령, 유시 섬에서 7km 떨어져 있는 섬의 스프링 발판의 세기가 3이라면, 스프링 발판을 딛고 도달할 수 있는 섬은 유시 섬에서 4km 이상 10km 이하 떨어져 있는 섬들이다. 다리오는 공주를 구하기 위해 앞만 보고 질주한다. 공주를 구하기 전에는 스프링 발판을 밟고 후퍼 섬을 향해서만 점프한다.

하지만 공주를 구한 뒤에는 공주를 들쳐 업고 유시 섬을 향해서만 뒤도 돌아보지 않고 도망친다. 일부 스프링 발판은 내구도가 너무 약해서 공주를 들쳐 업은 상태에서는 발만 딛어도 부서져버리기도 한다.

그림에서 유시 섬에서 12km 떨어진 곳에 있는 회색으로 표시된 섬의 스프링 발판이 그 예이다.

공주 구하기(L) (계속)

이런 스프링 발판들은 공주를 구하러 후퍼 섬을 향해 갈 때에만 사용할 수 있다.

유시 섬과 후퍼 섬을 포함한 모든 섬들의 정보와 섬마다 하나씩 있는 스프링 발판의 정보가 주어질 때, 다리오가 유시 섬을 출발해 공주를 구하고 돌아오는 서로 다른 경로의 개수를 1000으로 나눈 나머지를 출력하는 프로그램을 작성하시오.

입력

첫째 줄에는 섬의 개수 n이 주어진다. n은 유시 섬과 후퍼 섬도 포함한다.

(3 <= n <= 20)

이어지는 n개의 줄에는 각각의 섬에 대한 정보가 한 줄에 하나씩 주어진다. 섬의 정보는 유시 섬과의 거리가 가까운 순으로 주어진다. 그러므로 첫 번째로 정보가 주어지는 섬은 항상 유시 섬이고, 마지막으로 정보가 주어지는 섬은 항상 후퍼 섬이다.

섬의 정보를 나타내는 각각의 줄에는 섬에 대한 정보를 표현하는 세 개의 정수가 빈칸을 사이에 두고 주어진다. 첫 번째 정수 D는 유시 섬과의 거리이다. 유시 섬에 대해서는 D는 0이고, 후퍼 섬에서 D값이 가장 크다. D값이 동일한 두 섬은 존재하지 않는다.

두 번째 정수 S는 스프링 발판의 세기, 즉 해당 섬에서 좌우로 얼마나 떨어진 섬까지 점프할 수 있는지를 나타내는 값이다. 세 번째 정수 A는 해당 섬의 스프링 발판을 오렌지 공주를 들쳐 업은 상태에서도 사용할 수 있는지를 나타내는 값이다.

1이면 오렌지 공주와 함께 이용할 수 있고, 0이면 이용할 수 없다. 후퍼 섬에서 이 값은 항상 1이다.

출력

첫째 줄에 유시 섬에서 출발해 오렌지 공주를 구해오는 총 경로의 수를 1,000으로 나눈 나머지를 출력한다.

입력 예	출력 예
8 0 7 1 3 4 1 6 8 1 8 6 1 12 2 0 13 2 1 14 2 1 15 7 1	6

출처: 한국정보올림피아드(2008 지역본선 고등부)

풀이

이 문제는 중급편에서 전체탐색으로 도전했던 문제이다. 하지만 전체탐색법은 너무 많은 시간이 걸려 모든 케이스에 대해서는 해결할 수 없었다. 이 문제에 대해서 간단하게 설명하면 다음과 같다.

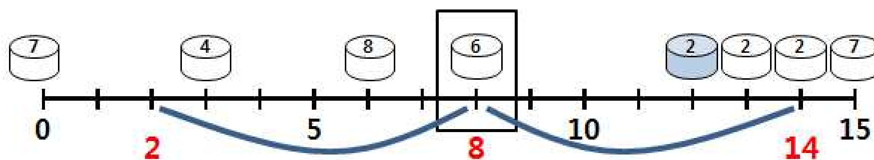
처음 위치에서 출발해서 마지막 위치에 도착한 후, 다시 처음 위치로 돌아오는 모든 경우의 수를 계산하고 그 결과를 이용해 답을 출력해야 하는 문제이다.

단, 어떤 발판을 밟게 되면 최대로 도달 가능한 다음 발판까지의 거리가 주어지며, 한 번 사용한 발판은 다시 사용하지 못한다.

또한, 마지막 위치에 도달한 후 다시 처음 위치로 돌아올 때에는 사용할 수 없는 특별한 발판이 존재한다.

문제에서는 마지막 위치까지 도달할 수 있는 경우를, 각 발판에서 점프할 수 있는 최대 크기와 오는데 사용할 수 없는 발판 정보를 이용해 효과적으로 문제 해결 방법을 찾아내는 것이 핵심이 된다.

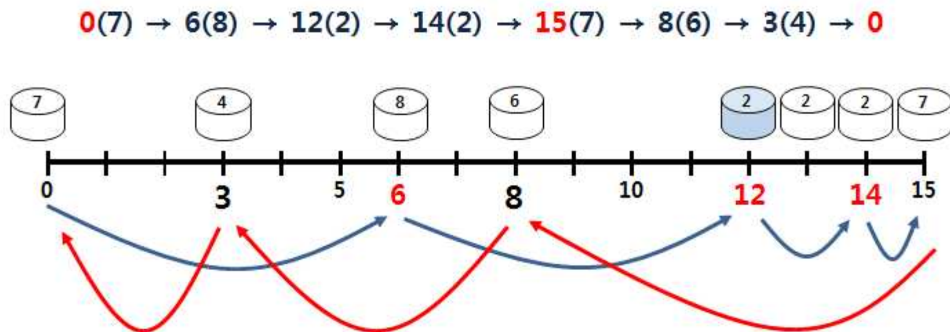
예를 들어 8번 위치에 있는 섬의 발판을 이용하는 경우, 그 발판의 강도가 6이므로 다음 섬은 2km이상 14km이하 떨어져 있는 섬들 중 하나로 이동할 수 있다.



즉, 8번 위치에 있는 발판을 밟는 경우 3, 6, 12, 13, 14번 섬으로 이동 가능하다.

처음 0위치에서 7만큼 이동 가능한 발판은 3, 6에 위치가 가능하고, 만약 6위치의 발판을 선택하게 되면 그 다음에 8만큼 이동 가능한 8, 12, 13, 14 위치의 한 섬을 선택할 수 있다.

위와 같은 과정을 반복적으로 실행해, 원래의 처음 위치로 돌아오는 모든 경우를 찾아야 한다. 예를 들어 첫 번째 섬에서 출발해 마지막 섬에서 공주를 구해 돌아오는 한 가지 경로에 대해서, "발판위치(발판강도)"를 이용해 아래와 같이 표현할 수 있다.



이 문제를 해결하는 기본적인 아이디어는 다리오가 유시 섬 – 후퍼 섬 – 유시 섬으로 왕복하는 것으로 해결해야 하지만 이렇게 접근하는 것은 문제를 해결하기 쉽지 않다. 따라서 다리오가 2명 있다고 생각하고 동시에 유시 섬에서 출발하여 후퍼 섬까지 가는 문제로 바꿔서 해결하는 것이 훨씬 유리하다.

이렇게 움직일 때, 하나의 다리오는 유시 섬에서 후퍼 섬으로 이동한다고 생각하고, 다른 다리오는 후퍼 섬에서 유시 섬으로 이동하는 것으로 가정해야 하기 때문에 점프의 거리 계산 등에 신경을 써야 한다.

먼저 위 아이디어를 반영한 동적 백트래킹 기법을 이용하기 위하여 정수형 값을 반환하는 탐색함수 f를 설계하자.

> $f(a,\ b,\ k)$ ="가는 다리오가 a 위치의 섬에 있고, 오는 다리오가 b 위치의 섬에 있으며, 다음으로 방문할 섬은 k 위치의 섬에 대해서 탐색하려고 하는 상태"

따라서 위 함수를 이용하여 전체탐색법으로 먼저 알고리즘을 작성한 후, 각 상태를 동적 표를 이용하여 저장하면 동적 백트래킹 기법으로 빠른 시간에 해결 가능한 알고리즘을 만들 수 있다.

이 방법으로 작성한 소스코드는 다음과 같다.

```
#include <cstdio>
#include <cstring>
#define MOD 1000

int n, D[501], S[501], A[501];
int DT[501][501];

int f(int a, int b, int k)
{
  if(DT[a][b]==-1)
  {
    if(k==n-1) return ((D[a]+S[a]>=D[k]) && (D[b]+S[k]>=D[k]));
    DT[a][b]=f(a,b,k+1);
    if(D[a]+S[a]>=D[k]) DT[a][b]+=f(k,b,k+1);
    if(D[b]+S[k]>=D[k] && A[k]) DT[a][b]+=f(a,k,k+1);
    DT[a][b]%=MOD;
  }
  return DT[a][b];
}

int main()
{
  scanf("%d", &n);
  memset(DT, -1, sizeof(DT));
  for(int i=0; i<n; i++)
    scanf("%d %d %d", D+i, S+i, A+i);
  printf("%d\n", f(0,0,1));
  return 0;
}
```

다음으로 이 문제를 동적계획법으로 설계하고 해결해보자. 동적계획법으로 해결하기 위해서는 먼저 DT를 다음과 같이 정의한다.

> $DT[a][b]$ = "가는 다리오는 a 위치의 섬에, 오는 다리오가 b 위치의 섬이라고 있을 수 있는 모든 경우의 수"

이제 DT에 대한 관계를 만들어보자. 먼저 $DT[0][0]$은 1이다. 즉, 처음 출발위치인 유시 섬에서 출발하여 유시 섬에서 끝나야하므로 가는 다리오, 오는 다리오 둘 모두 처음에는 유시 섬에 있는 경우가 1가지 있다. 다음으로 $DT[a][b]$에 대한 관계를 유도해보자.

먼저 가는 다리오가 이동하여 임의의 k번째 섬에서 a번째 섬으로 이동하는 경우는 $DT[a][b]$는 $DT[k][b]$인 상태들 중에서 $d_k + s_k \geq d_a$이면서, $k < a$인 상태들의 합으로 될 수 있다. 왜냐하면 가는 다리오가 k번째 섬에서 a번째 섬으로 점프할 수 있으면 모두 a칸으로 이동할 수 있기 때문이다.

다음으로 오는 다리오가 이동하여 임의의 k번째 섬에서 b번째 섬으로 이동하는 경우 $DT[a][b]$는 $DT[a][k]$들 중에서 $d_k + s_b \geq d_b$이면서 $k < b$, 그리고 $a_b = 1$(회색이 아닌 발판)인 상태들의 합으로 나타낼 수 있다. 여기서 다른 점은 돌아오는 다리오의 점프력은 실제로 방향을 거꾸로 생각해야하기 때문에 도착점에서 출발점으로 점프하는 것으로 생각해야 한다.

위의 내용을 식으로 정리하면 다음과 같다.

$$DT[a][b] = \begin{cases} 1 & (a = 0,\ b = 0) \\ \sum_{k=0}^{a-1} DT[k][b] & (a > b,\ d_k + s_k \geq d_a) \\ \sum_{k=0}^{b-1} DT[a][k] & (b > a,\ a_b = 1,\ d_k + s_b \geq d_b) \end{cases}$$

위 식에서 a와 b의 조건에 따라, 더 큰 값에 대하여 이동하는 것이므로, 다음과 같이 a와 b의 조건이 추가되었다. 위 식으로 구현한 소스코드는 다음과 같다.

```
#include <cstdio>
#define MOD 1000

int n, D[501], S[501], A[501], DT[501][501];

int main()
{
  scanf("%d",&n);
  for(int i=0; i<n; i++)
    scanf("%d %d %d", D+i, S+i, A+i);
  for(int i=0; i<n; i++)
  {
    for(int j=0; j<n; j++)
    {
      if(i==0 && j==0) DT[i][j]=1;
      else if(i!=j || i==n-1 && j==n-1)
      {
        if(i>j)
        {
          for(int k=; k<i; k++)
            if(D[k]+S[k]>=D[i]) DT[i][j]+=DT[k][j];
        }
        else if(A[j])
        {
          for(int k=0; k<j; k++)
            if(D[k]+S[j]>=D[j]) DT[i][j]+=DT[i][k];
        }
      }
      DT[i][j %=MOD;
    }
  }
  printf("%d\n", DT[n-1][n-1]);
  return 0;
}
```

또 다른 방법으로 $DT[i][j]$ 로부터 다음 k 지점을 향하여 진행하는 방법으로도 상향식 동적계획법을 구현할 수 있다. 이 방법은 소스코드를 분석하여 어떻게 구현한 것인지 연구해보기 바란다.

코드	참고

```
#include<cstdio>

int DT[500][500];
int D[500];
int S[500];
bool A[500];

int main()
{
  int i,j,k,n,ans=0;
  scanf("%d", &n);
  for(i=0; i<n; i++)scanf("%d %d %d",&D[i],&S[i],&A[i]);
  DT[0][0]=1;
  for(i=0; i<n-1; i++)
  {
    for(j=0; j<n-1; j++)
    {
      if((i==0 && j==0) || (i != j&&A[j]))
      {
        k=i;
        if(k<j) k=j;
        for(k++; k<n; k++)
        {
          if(S[i]>=D[k]-D[i])DT[k][j]=(DT[k][j]+DT[i][j])%1000;
          if(S[k]>=D[k]-D[j])DT[i][k]=(DT[i][k]+DT[i][j])%1000;
        }
      }
    }
  }
  for(i=0; i<n-1; i++) if(S[i]>=D[n-1]-D[i])
    ans=(ans+DT[i][n-1])%1000;
  printf("%d", ans);
  return 0;
}
```

3 동적표를 이용한 중급 기법

이번에는 보다 수준이 높은 동적계획법에 대해서 알아보자. 지금까지의 문제들은 상태가 일반적으로 하나의 정수 형태로 표현됐다. 하지만 상태를 정수로 표현할 수도 있지만 이를 bit를 이용하여 효율적으로 표현하고, 이를 통하여 동적표를 활용하는 방법이 있다.

일단 간단하게 비트로 상태를 표현하는 방법에 대해서 알아보자. 일반적으로 어떤 노드를 방문했는지 방문하지 않았는지를 체크할 때 bool형으로 배열을 활용하는 경우가 많다. 이런 경우, 다음과 같이 처리한다.

```
bool chk[SIZE];  //체크 배열의 준비
...
chk[k] = true      //k번째 상태를 방문했다고 체크
...
if( !chk[k] )     //k번째 상태를 방문하지 않았으면
```

기본적으로 위와 같은 형태로 배열을 활용하는 경우가 많다. 만약 위와 같은 배열을 활용할 때, SIZE가 30이하인 경우에는 정수형 변수 하나의 각 비트를 활용하여 상태를 표현할 수 있다. 그 예는 다음과 같다.

bool chk[SIZE]; chk[k] = true if(!chk[k])	int chk; chk = chk | (1<<k); if((~chk) & (1<<k))

위와 같이 표현할 경우 비트를 이용하므로 속도도 빠르고, 동적계획법에서의 상태 표현에서 효율적인 경우가 있다. 주어진 문제들을 통해, 상태를 비트로 표현하는 동적계획법에 대해서 익힐 수 있도록 하자.

문제 1

minimum sum(L)

n*n개의 수가 주어진다(1<=n<=20).

이때 겹치지 않는 각 열과 각 행에서 수를 하나씩 뽑는다.
(즉, 총 n개의 수를 뽑을 것이다. 그리고 각 수는 100 이하의 값이다.)

이 n개의 수의 합을 구할 때 최솟값을 구하시오.

입력

첫 줄에 n이 입력된다. 다음 줄 부터 n+1줄 까지 n개씩의 정수가 입력된다.

출력

구한 최소 합을 출력한다.

입력 예	출력 예
3 1 2 5 2 4 3 5 4 3	7

풀이

이 문제는 이미 중급 편에서 전체탐색법으로 해결한 적이 있는 문제이다. 하지만 전체탐색으로는 맵의 크기가 10을 초과하게 되면 너무 많은 시간이 걸려서 해결하기 쉽지 않다. 하지만 여기서는 20정도의 값일 때에도 해결할 수 있는 방법을 소개한다.

중급 편에서는 한 행의 왼쪽에서부터 하나씩 원소를 선택하고 선택한 열을 chk배열에 체크함으로써 다음 행에서 이미 이전에 체크된 열을 선택하지 않도록 탐색을 진행하여 $n!$ 정도의 계산량으로 문제를 해결했다.

여기서 계산량을 줄일 수 있는 아이디어를 생각해보자. 다음과 같은 입력이 주어진다고 가정하자.

5	2	3	7	1
6	2	9	4	2
1	2	3	4	5
9	8	1	5	9
1	3	2	5	7

첫 행, 첫 열의 값을 선택한다.

5	2	3	7	1
6	2	9	4	2
1	2	3	4	5
9	8	1	5	9
1	3	2	5	7

다음으로 첫 열의 값은 선택할 수 없으므로 다음으로 2열의 값을 선택할 수 있다.

5	2	3	7	1
6	**2**	9	4	2
1	2	3	4	5
9	8	1	5	9
1	3	2	5	7

다음 열에서는 만약 임의로 5번째 열을 선택했다고 가정하자.

5	2	3	7	1
6	**2**	9	4	2
1	2	3	4	**5**
9	8	1	5	9
1	3	2	5	7

이 상태에서 앞으로 남은 2행에 대해서만 생각해보자. 이때는 고를 수 있는 값은 아래 진한 색 부분의 값이 중복되지 않도록 1, 5 또는 5, 2뿐이다. 물론 이 상태라면 무조건 1, 5를 선택해야 한다. 왜냐하면 최솟값을 고르는 문제이기 때문이다.

그렇다면 위 그림의 상태와 같은 다른 상태는 또 존재할까? 물론 존재한다. 다음 제시하는 모든 상태는 위 그림과 동일한 상태로 무조건 1, 5를 고르는 것이 이득이다.

5	2	3	7	1
6	**2**	9	4	2
1	2	3	4	**5**
9	8	1	5	9
1	3	2	5	7

5	2	3	7	1
6	2	9	4	**2**
1	**2**	3	4	5
9	8	1	5	9
1	3	2	5	7

5	**2**	3	7	1
6	2	9	4	2
1	2	3	4	**5**
9	8	1	5	9
1	3	2	5	7

5	**2**	3	7	1
6	2	9	4	**2**
1	2	3	4	5
9	8	1	5	9
1	3	2	5	7

5	2	3	7	**1**
6	2	9	4	2
1	**2**	3	4	5
9	8	1	5	9
1	3	2	5	7

5	2	3	7	**1**
6	**2**	9	4	2
1	2	3	4	5
9	8	1	5	9
1	3	2	5	7

위에 제시된 6개는 모두 다른 상태이지만 4행과 5행에 대해서 고르는 값은 모두 동일하다. 따라서 현재까지의 상태를 같은 상태로 표현할 수 있다.

위 6개의 값의 공통점은 무엇인가? 조금만 관찰하면 쉽게 발견할 수 있다. 위 값들은 모두 체크 배열의 값이 1, 2, 5행에 체크된 값들이다. 즉, 체크배열의 체크된 값이 같다면 나머지 남은 행에 대한 선택은 동일함을 알 수 있다.

만약 체크 배열을 하나의 정수값으로 표현한다면 하나의 상태로 나타낼 수 있으며, 이 값을 이용한 관계식을 유도하여 동적표를 활용할 수 있다.

이 문제에서 n의 값이 20이므로 2^{20}으로 $DT[2^{20}]$을 선언하여 서로 다른 상태를 저장할 수 있다.

중급편에서 작성했던 함수를 이용하여 동적표를 활용해 보자. 중급편의 void 함수를 동적표를 활용할 수 있도록 정수형 함수로 바꾼 소스코드는 다음과 같다.

줄 | 코드 | 참고

```
#include <stdio.h>
#define INF 987654321

int m[21][21], col_check[21], n;

void input()
{
  scanf("%d",&n);
  for(int i=0 ; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &m[i][j]);
}

int min(int a, int b) { return a>b ? b:a;}

int solve(int row)
{
  int ans=INF;
  if(row==n) return 0;
  for(int i=0; i<n; i++)
  {
    if(col_check[i]==0)
    {
      col_check[i]=1;
      ans=min(ans, solve(row+1)+m[row][i]);
      col_check[i]=0;
    }
  }
  return ans;
}

int main()
{
  input();
  printf("%d", solve(0));
  return 0;
}
```

여기서 동적표를 활용하기 위해서 col_check[21]의 내용을 21bit의 정수로 만들어서 동적표를 적용해보자. 이 방법은 매우 효율적으로 활용할 수 있으므로 반드시 익혀둘 필요가 있다.

먼저 체크 배열에 체크하는 방법과 체크되었는지 검사하는 방법에 대해서 다음과 같이 bit로 표현할 수 있다.

<table>
<tr><td>col_check[i] == 0</td><td>(bit & (1<<i)) == 0</td></tr>
<tr><td>col_check[i] = 1</td><td>bit = bit | (1<<i) 또는 bit += (1<<i)</td></tr>
<tr><td>col_check[i] = 0</td><td>bit = bit - (1<<i)</td></tr>
</table>

이 방법을 이용하여 체크 배열을 bit를 이용하여 표현하면 다음과 같다.

코드	참고

```
#include <stdio.h>
#define INF 987654321

int m[21][21], bit, n;

void input()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &m[i][j]);
}

int min(int a, int b) { return a>b ? b:a;}

int f(int row)
{
  int ans=INF;
  if(row==n) return 0;
  for(int i=0; i<n; i++)
  {
    if((bit&(1<<i))==0)
    {
      bit+=(1<<i);
      ans=min(ans, f(row+1)+m[row][i]);
```

코드	참고

```
      bit-=(1<<i);
    }
  }
  return ans;
}

int main()
{
  input();
  printf("%d", f(0));
  return 0;
}
```

이제 이 값을 이용하여 동적표에 적용해보자.

$DT[bit] = f(row, bit)$의 값을 기록해 둔다.

코드	참고

```
#include<stdio.h>
#define INF 987654321

int m[21][21], bit, n;
int DT[1<<20];

void input()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &m[i][j]);
}

int min(int a, int b) { return a>b ? b:a;}

int f(int row, int bit)
{
  if(row==n) return 0;
  if(DT[bit]==0)
  {
    DT[bit]=INF;
```

```
for(int i=0; i<n; i++)
      if((bit&(1<<i))==0)
        DT[bit]=min(DT[bit], f(row+1,bit+(1<<i))+ m[row][i]);
  }
  return DT[bit];
}

main()
{
  input();
  printf("%d", f(0,0));
  return 0;
}
```

이와 같이 bit를 이용하여 동적표에 저장하여 효율을 향상시킬 수 있다. 이 경우에는 n의 값이 20이더라도 제한된 시간 이내에 해를 구할 수 있다. 이 방법을 적절히 활용하면 상향식 동적계획법으로도 작성할 수 있으므로 이 알고리즘이 완벽히 이해가 된다면 도전해보기 바란다.

문제 2

두부모판 자르기

KOI 두부 공장에서 만들어내는 크기가 n*n(n은 11 이하의 자연수)인 두부 모판이 있다. 이 모판을 1*1 크기의 단위두부가 2개 붙어있는 형태의 포장단위(즉, 1*2 혹은 2*1 크기)로 잘라서 판매한다.

그런데 두부제조 공정상 모판에 있는 각 단위두부의 품질은 A, B, C, F급으로 분류되고, 잘려진 포장단위의 두부 가격은 이 포장단위에 있는 두 개의 단위두부의 품질에 따라서 그림 1과 같이 정해진다.

등급 \ 등급	A	B	C	F
A	100	70	40	0
B	70	50	30	0
C	40	30	20	0
F	0	0	0	0

그림 1. 포장단위의 가격표

포장단위에 있는 두 단위두부가 [A,A]급이면 100원을 받고, [A,B]급이면 70원을, [A,C]급이면 40원을, [B,B]급이면 50원을, [B,C]급이면 30원을, [C,C]급이면 20원을 받는다.

포장단위에 있는 두 개의 단위두부 중 하나라도 F급이 있으면 이 포장단위는 한푼도 받을 수 없다. 두부 모판의 품질이 주어질 때, 가장 높은 가격을 받도록 두부 모판을 1*2 혹은 2*1 크기의 포장단위들로 자르고자 한다. 예를 들어 그림 2와 같은 3*3 두부 모판이 주어져 있다고 하자.

두부모판 자르기 (계속)

A	A	C
F	C	A
A	C	B

그림 2. 포두부 모판의 예

A	A	C
F	C	A
A	C	B

그림 3. 포잘려 진 두부모판

이 경우, 그림 3과 같이 자르면 4개의 포장단위가 만들어진다. 이때, 이들 포장단위의 가격은 [A,A]=100, [F,C]=0, [A,C]=40, 그리고 [A,B]=70 이다.

여기서 오른쪽 위 [C]와 같이 단위두부 하나는 포장단위가 아니므로 판매할 수 없다. 따라서 총 가격은 210원이 된다. 이 가격은 그림 2와 같은 3*3 두부모판에서 받을 수 있는 가장 높은 가격이다.

두부모판의 크기와 단위두부의 등급이 주어질 때, 이를 포장단위로 잘라 받을 수 있는 총 가격의 최댓값을 구하는 프로그램을 작성하시오.

입력

첫째 줄에는 두부모판의 크기를 나타내는 n(2<= n <= 11)이 주어진다. 둘째 줄부터 n줄에 걸쳐 각 줄에 두부모판의 단위두부의 등급들이 행단위로 등급사이에 공백 없이 첫 번째 행부터 차례로 주어진다.

출력

입력된 두부모판을 포장단위로 잘라서 받을 수 있는 최대 가격을 첫째 줄에 출력한다.

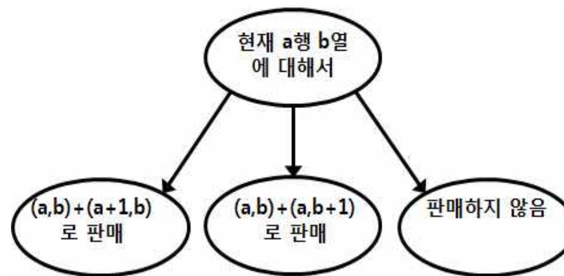
입력 예	출력 예
3 AAC FCA ACB	210

출처: 한국정보올림피아드(2006 전국본선 고등부)

풀이

이 문제는 한국정보올림피아드에 출제되었던 당시 대부분의 학생들이 해결하지 못한 문제로 그 당시로는 난이도가 상당히 높았던 문제이다. 이 문제는 다른 방법으로도 해결할 수 있지만 여기서는 bit구조를 이용한 동적표를 이용하여 해결해 본다.

이 문제를 bit구조를 이용하여 해결하기 위해서는 먼저 전체탐색법 중 한 방법인 백트래킹 기법을 이용하여 해결할 수 있도록 탐색을 구조화해야한다. 먼저 탐색구조를 다음과 같이 구성해보자.



위와 같이 현재 상태에서 3가지 상태를 생성해가면서 모든 상태를 탐색한 후 가장 이득이 되는 값을 출력하면 해가 된다.

하지만 이 방법은 $3^{(11\times11)}$의 계산량을 가지므로 제한된 시간에 해를 구할 수 없지만, 동적표를 활용하면 충분히 제한된 시간에 해를 구할 수 있는 알고리즘으로 효율을 높일 수 있다.

먼저 전체탐색법으로 구현한 소스코드는 다음과 같다. 이 소스코드의 작성도 쉬운 것이 아니므로 잘 익혀둘 수 있도록 한다.

줄 | 코드 | 참고

```
#include <cstdio>

int chk[20][20],P[26][26]={{100,70,40},{70,50,30},{40,30,20}}, n;
char A[20][20];

int max(int a, int b) { return a>b ? a:b;}
```

```

int f(int a, int b)
{
  int ans=0;
  if(a==n) return 0;
  if(b==n) return f(a+1, 0);
  if(chk[a][b]==0)
  {
    chk[a][b]=1;
    if(b+1<n && chk[a][b+1]==0)
    {
      chk[a][b+1]=1;
      ans=max(ans,f(a,b+1)+ P[A[a][b]-'A'][A[a][b+1]-'A']);
      chk[a][b+1]=0;
    }
    if(a+1<n && chk[a+1][b]==0)
    {
      chk[a+1][b]=1;
      ans=max(ans,f(a,b+1)+ P[A[a][b]-'A'][A[a+1][b]-'A']);
      chk[a+1][b]=0;
    }
    ans=max(ans, f(a,b+1));
    chk[a][b]=0;
  }
  else
    ans=max(ans, f(a,b+1));
  return ans;
}

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%s", &A[i]);
  printf("%d\n", f(0, 0));
  return 0;
}
```

이 알고리즘은 해는 정확하게 출력하나, 시간이 너무 오래 걸려서 제한 시간 내에 해를 구할 수 없다. 하지만 여기서도 앞에서 나왔던 문제와 마찬가지로 chk배열의 내용을 bit로

처리하면 동적표를 이용할 수 있다.

하지만 이 문제에 있어서는 테이블의 크기가 11×11이라서 하나의 정수를 이용하여 bit를 표현하려면 2^{121}의 정수가 필요하기 때문에 활용할 수가 없다.

그렇지만 이 문제의 chk배열의 최적화기법을 통하여 bit의 크기를 확실하게 줄일 수 있다. 다음 그림을 통해서 chk배열을 최적화해보자.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

만약 체크 배열이 위와 같은 상태로 채워져 있으며, 진하게 표시된 칸을 탐색할 차례라고 생각해 보자.

이 경우 모든 테이블의 값을 기억하고 있을 필요가 없음을 알 수 있다. 지금 상태에서 검색하려는 칸은 다음 2가지 경우만 처리하면 된다.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

현재 상황에서 진한 테두리로 된 현재 탐색 영역의 윗부분은 1로 채워져 있음이 명확하다. 그리고 어떤 칸이 1이라면 그 아래쪽 칸은 1일 가능성이 있는 칸들이다.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

회색으로 표시된 칸 들은 위의 1이 표시된 부분 중 테두리가 진한 선으로 표시된 부분들에 의해서 1일 수도 있고 0일 수도 있기 때문에 값을 확정할 수 없는 칸들이다.

그리고 지금 탐색을 하려는 칸의 값에 의해 그 아래 값도 1이 될 가능성이 있는 칸으로 볼 수 있다.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?
?	?	?	?	?	?	?

따라서 지금 회색으로 표시된 칸들 이외에 다른 값들은 모두 값이 확정되어 있음을 쉽게 알 수 있다. 즉 지금 회색이 표시되지 않은 아래쪽 "?"들은 모두 아직 탐색되지 않은 부분임이 명확하므로 값이 0임이 확실하다.

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	?	?	?	?	?
?	?	?	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

위 그림과 같이 값들이 배치됨을 알 수 있고, 현재 탐색 중인 칸은 "?"이지만 현재 0일 경우만 탐색을 진행하므로 실제로 chk배열로 값을 확정하지 못해서 기억할 부분은 연한 회색으로 표시된 n개의 칸들의 값들만 기억하며, 다음 상태로 갈 때 bit-state값을 shift시키면서 관리하면 된다. 이를 bit-스크롤링 기법이라고도 한다.

따라서 회색으로 표시된 칸들만 bit로 구성하여 기억한다면 이 문제의 경우 2^{11}의 크기와 현재 탐색 가능한 칸의 위치를 나타내는 행과 열 즉 11×11의 공간만 있으면 모든 상태를 표현할 수 있다.

따라서 충분히 동적표로 처리할 수 있음을 알 수 있다. 이와 같이 처리할 수 있는 문제들이 가끔 등장하므로 이 기법을 반드시 익혀둘 수 있도록 한다. 따라서 앞에서 해결했던 탐

색 함수에 현재 상태를 하나 더 표현한 함수를 다음과 같이 정의할 수 있다.

$f(a,\ b,\ bit-state)$ ="현재 a행 b열을 탐색하는 상태에서 $bit-state$의 값으로 다음 n개의 칸이 채워져 있을 때까지의 최대 판매 이득"

그리고 이 탐색함수를 이용할 때에는 chk배열을 사용할 필요가 없다. 그리고 이 탐색함수를 동적테이블 DT에 다음과 같이 채울 수 있도록 구성한다.

$DT[a][b][bit_{state}] \ = \ f(a,\ b,\ bit-state)$ 의 값으로 채운다.

이와 같이 처리하면 계산량은 $n^2 2^n$이 되므로 $n=11$인 상태에서 충분히 해결 가능한 문제가 된다. 이를 적용한 소스코드는 다음과 같다.

줄	코드	참고

```
#include <stdio.h>
#define cu (1<<n)
#define rt (1<<(n-1))
#define dn (1)
#define M (1<<(n+1))

int p[4][4]={100,70,40,0,70,50,30,0,40,30,20,0,0,0,0,0};
int n, tb[12][12], m[12][12][1<<13];

int max(int a, int b) { return a>b ? a:b;}
int f(int x, int y, int s)
{
  if(x==n) return 0;
  if(y==n) return f(x+1, 0, s);
  if(!m[x][y][s])
  {
    if(!(s&cu))
    {
      if(y+1<n && !(s&rt))
        m[x][y][s]=max(m[x][y][s], f(x,y+2,(s<<2)%M)+p[tb[x][y]][tb[x][y+1]]);
      if(x+1<n && !(s&dn))
        m[x][y][s]=max(m[x][y][s], f(x,y+1,((s|dn)<<1)%M)+p[tb[x][y]][tb[x+1][y]]);
      m[x][y][s]=max(m[x][y][s],f(x,y+1,(s<<1)%M));
    }
    else m[x][y][s]=max(m[x][y][s],f(x,y+1,(s<<1)%M));
```

줄	코드	참고
26	`  }`	
27	`  return m[x][y][s];`	
28	`}`	
29		
30	`int main()`	
31	`{`	
32	`  int i, j; char t;`	
33	`  scanf("%d",&n);`	
34	`  for(i=0; i<n; i++)`	
35	`    for(int j=0; j<n; j++)`	
36	`    {`	
37	`      scanf(" %1c", &t);`	
38	`        tb[i][j]=(t=='F' ? 3:t-'A');`	
39	`    }`	
40	`  printf("%d\n",f(0,0,0));`	
41	`  return 0;`	
42	`}`	
43		

** 20: m[x][y][s]=max(m[x][y][s], f(x,y+2,(s<<2)%M)+p[tb[x][y]][tb[x][y+1]]);

** 22: m[x][y][s]=max(m[x][y][s], f(x,y+1,((s|dn)<<1)%M)+p[tb[x][y]][tb[x+1][y]]);

매우 다양한 비트처리 기법들이 한꺼번에 활용되어 소스코드의 이해가 쉽지 않지만 차근차근 분석해 보면 이해할 수 있으며, 이러한 코드를 실전에서도 작성할 수 있을 것이다. 충분히 활용할 수 있도록 익혀두자.

물론 이 문제 또한 상향식 동적계획법으로도 작성 가능하므로, 위 소스코드를 이해한 다음 상향식 동적계획법으로도 작성할 수 있도록 연습해보자.

Part

II

알고리즘 설계기법의 응용

4. 이분탐색을 활용한 설계기법
5. 자료구조를 활용한 알고리즘의 고속화

4 이분탐색을 활용한 설계기법

문제를 해결하는데 있어서 이분탐색은 매우 효율적인 해결방법이다. 일반적으로 최솟값의 최대화, 최댓값의 최소화 등의 문제에 이용할 수 있는 경우가 많다.

이분탐색으로 문제를 해결하는 경우에 일반적으로 최적화 문제를 결정 문제로 바꿔서 생각한다. 따라서 결정 문제의 참/거짓의 결과를 이용하여 이분탐색으로 가장 최적의 해를 탐색하는 방법으로 해를 구한다. 먼저 다음 예제를 통하여 기본적인 접근법에 대해서 알아보자.

예

KOI왕국의 임금은 퀴즈를 매우 좋아한다. 이 임금은 세상에서 가장 퀴즈를 잘 푸는 당신에게 퀴즈의 정답을 맞히면 상금을 지불하려고 한다. 임금은 상금의 액수를 마음속으로 이미 결정하고 있다. 당신에게 주어진 행위는 다음과 같은 질문을 할 수 있다.

“제가 상금으로 x 원을 받아갈 수 있습니까?”

위 질문에서 x 값을 결정할 수 있다. 이 때 임금의 대답은 Yes 혹은 No이다. 임금은 자신이 주려는 상금보다 적은 액수에 대해서는 지불할 의사를 가지고 있기 때문에, Yes라고 대답한 경우에는 지불할 액수보다 같거나 적다는 의미이다. 만약 질문의 x 가 임금이 지불하려는 액수보다 큰 금액을 요구하면 No라고 대답한다.
당신이 받을 수 있는 상금의 최대 금액은 얼마인가? 즉, 임금이 처음 지불하려고 생각했던 금액이 받아가려고 했던 금액이기 때문이다.
단, 임금이 상금으로 생각하는 금액의 범위는 2^{30} 보다는 작은 자연수 값이다.

풀이

이 문제는 일단 쉽게 해결할 수 있는 방법이 범위에 해당하는 모든 값을 하나씩 대입해보는 방법이다.

줄 / 코드 / 참고

```
1   #include <cstdio>
2   #define SOL 998987
3
4   bool f(int x)
5   {
6     return (998987>=x);
7   }
8
9   int main()
10  {
11    for(int i=1; i<=1<<30; i++)
12    if(!f(i))
13    {
14      printf("%d\n", i-1);
15      return 0;
16    }
17    return 0;
18  }
```

참고: 2: 임금이 정한 금액

이 문제는 주어진 문제를 결정 문제로 바꿔서 생각한다. 결정문제 "x 원의 상금을 받을 수 있는가?"의 결과에 따라서 해를 구할 수 있다.

이 풀이에서는 x를 1부터 차례로 증가시키면서 불가능하게 되는 순간 그 이전 값이 가장 큰 액수임을 이용한 전체탐색 풀이이다. 이 방법은 너무 느리기 때문에 경우에 따라서는 해를 구할 수 없을 수 있다.

다음의 이분탐색 전략으로 접근해보자. 기본적인 방법은 다음과 같다.

① lb와 ub를 1과 2^30으로 설정한다.
② lb 〈 ub일 동안 반복한다. 이 조건이 거짓이면 ⑤로 이동
③ m = (lb + ub + 1)/2로 설정한다.
④ 만약 m의 상금을 받을 수 있다면 lb = m으로 설정하고 ②번으로 이동, 아니면 ub = m-1로 설정하고 계속 탐색
⑤ ub를 출력한다.

줄	코드	참고
1	`#include <cstdio>`	2: 임금이 정한 금액
2	`#define SOL 998987`	
3		
4	`bool f(int x)`	
5	`{`	
6	`  return (SOL >= x);`	
7	`}`	
8		
9	`int main()`	
10	`{`	
11	`  int lb=0, ub=1<<30, m;`	
12	`  while(lb<ub)`	
13	`  {`	
14	`    m=(lb+ub+1)/2;`	
15	`    if(f(m)) lb=m;`	
16	`    else ub=m-1;`	
17	`  }`	
18	`  printf("%d\n", ub);`	
19	`  return 0;`	
20	`}`	

이 방법은 매우 다양한 용도로 활용할 수 있기 때문에 잘 익혀둘 수 있도록 한다. 이와 관련 있는 이분탐색 법은 lower bound, upper bound 등이 있으며, 이는 중급 편에서 다루고 있으므로 한 번 다시 볼 수 있도록 한다. 주어진 문제들을 통하여 이분탐색을 활용한 문제해결을 익힐 수 있도록 하자.

문제 1

제자리멀리뛰기

KOI고등학교에서는 체력 측정에서 제자리멀리뛰기가 가장 중요하다. KOI의 체육선생님께서는 학생들의 제자리멀리뛰기 실력을 키우기 위해 특수 훈련을 준비 중이다.

특수 훈련 장소는 KOI고등학교 특수 트레이닝 센터로 이 곳은 끓는 용암으로 가득차 있다. 체육 선생님께서는 이 용암으로 가득찬 방의 가운데 있는 돌섬에 학생들을 가두고 학생들이 탈출해 나오기를 기대하고 있다. 탈출할 수 있는 방법은 단 한 가지뿐이다.

돌섬에서 탈출구까지 띄엄띄엄 존재하는 작은 돌섬들로 점프하여 탈출구까지 가는 것이다.

돌섬에서 탈출구 사이에는 총 n개의 작은 돌섬이 있다. 선생님은 이 n개의 작은 돌섬들 중 m개를 제거하여 학생들이 최대한 멀리뛰기 연습의 효율을 높이고, 학생들이 각 돌섬을 점프한 거리의 최솟값을 최대한 크게 하려고 한다.

물론 학생들은 체력이 좋기 때문에 두 돌섬이 아무리 멀더라도 점프할 수 있다. 즉, 빠지는 일은 없다.

그리고 학생들은 탈출 시 n-m개의 모든 돌섬을 밟으면서 탈출해야 한다.

학생들이 갇힌 돌섬으로부터 탈출구까지의 거리 d가 주어지고, 각 n개의 작은 돌섬의 위치(갇힌 돌섬으로 부터의 거리)가 주어지며, 제거할 수 있는 작은 돌섬의 수 m이 주어질 때, m개를 제거한 후 학생들이 점프하는 최소거리의 최댓값을 구하는 프로그램을 작성하시오.

제자리멀리뛰기 (계속)

입력

첫 번째 줄에는 갇힌 돌섬으로부터 탈출구까지의 거리 d(1<= d<=1,000,000,000), 작은 돌섬의 수 n(0<=n<=50,000), 제거할 수 있는 작은 돌섬의 수 m (0<=m<=n)이 공백으로 구분되어 주어진다.

두 번째 줄부터 m줄에 걸쳐서 갇힌 섬으로부터 각 작은 돌섬이 얼마나 떨어져 있는지를 나타내는 하나의 정수가 한 줄에 하나씩 주어진다(단, 두 돌섬은 같은 위치에 있을 수 없다.).

출력

m개의 작은 섬을 제거한 뒤 학생들이 점프할 수 있는 최소거리의 최댓값을 출력한다.

입력 예	출력 예
25 5 2 2 14 11 21 17	4

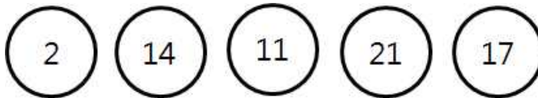
풀이

이 문제도 특별한 아이디어가 잘 떠오르지 않는 문제이다. 따라서 이 문제도 결정 문제를 해결한 후 이 결과를 이분탐색을 이용하여 해결해보자.

따라서 다음과 같은 결정 문제를 해결할 수 있는 효율적인 방법을 생각해보자.

거리 x 미만으로는 점프하지 않도록 m개의 돌을 제거할 수 있는가?

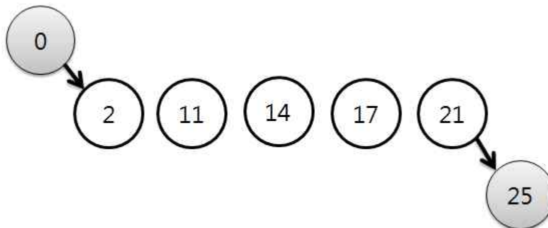
이 문제를 해결하기 위하여 먼저 주어진 돌섬의 위치들을 오름차순으로 정렬한다.



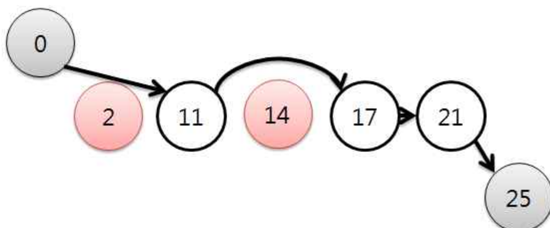
초기에 입력된 자료의 순서는 다음과 같다.



먼저 위치에 따라 오름차순으로 정렬한다.



입력 값이 0, 탈출구의 위치가 25이고 다음과 같은 형태가 된다.



왼쪽 그림과 같이 2와 14의 2개의 돌을 제거하면 4미만으로는 점프하지 않고 끝까지 가야 한다.

이 보다 더 큰 값은 불가능하다.

이와 같은 방법으로 문제를 해결하기 위해서는 먼저 최소 점프거리 x를 결정하고 다음 결정 문제를 해결하기 위하여 다음과 같은 단계로 진행한다.

① lb와 ub를 1과 1,000,000,000으로 설정한다.
② lb 〈 ub일 동안 반복한다. 이 조건이 거짓이면 ⑥으로 이동
③ L = (lb + ub + 1)/2로 설정한다.
④ 만약 현재 지점에서 거리 L이내에 다른 돌이 존재하면 돌을 모두 제거한다.
⑤ 탈출 지점까지 m개 이하의 돌을 제거하여 이동 가능하다면 lb = L로 설정 이동이 불가능하다면 ub = L-1로 설정하여 ②번으로 이동
⑥ ub를 출력한다.

이 단계를 소스코드로 구현한 결과는 다음과 같다.

코드	참고

```
#include <cstdio>
#include <algorithm>

int n, d, m, S[50001];
int ub = 1000000000, lb, L;

bool f(int x)
{
  int cnt=0, cur=S[0];
  for(int i=1; i<n+2; i++)
  {
    if(S[i]<cur+x) cnt++;
    else cur=S[i];
  }
  return (cnt<=m && cur==d);
}

int main()
{
  scanf("%d %d %d", &d, &n, &m);
  for(int i=0; i<n; i++)
    scanf("%d",S+i);
  S[n]=0, S[n+1]=d;
  std::sort(S, S+n+2);
  while(lb<ub)
  {
```

```
    L=(lb+ub+1)/2;
    if(f(L)) lb=L;
    else ub=L-1;
  }
  printf("%d\n", ub);
  return 0;
}
```

문제 2

암벽등반

정올이는 암벽등반을 시도하려고 한다. 암벽등반을 하기 위해서는 체력을 길러두어야 한다.

등산 장소는 각 장소의 돌출 정도가 표시되며 많이 튀어나와 있을수록 그 값이 크다. 각 지점의 돌출 정도는 0~1,000,000 중 하나의 값을 가진다.

체력은 H로 표현되며, 체력이 H라면 현재 위치에서 ±H이하의 상, 하, 좌, 우 장소로 이동할 수 있다. 따라서 체력이 높을수록 어려운 암벽등반에서 완주할 가능성이 높아진다.

정올이가 이번에 참가할 암벽등반 대회의 등산로의 정보가 지도로 주어진다. 지도는 N*N으로 구성된 정사각형이다. 암벽등반의 성공 여부는 주어진 지도의 3/4 이상의 장소를 지나가야 성공으로 판정한다.

정올이는 임의의 장소에서 출발할 수 있고, 끝나는 장소도 임의의 장소에서 끝낼 수 있다. 다만, 전체 등반 장소의 3/4이상을 이동해야 한다. 정올이가 주어진 지도의 암벽등반을 무사히 완주하기 위해서 필요한 최소 H를 구하는 프로그램을 작성하시오.

단, 3/4가 실수일 경우에는 소수점 이하를 버린 값만큼만 등반하면 성공으로 본다. 만약 N이 9라면 모두 81개의 등반 장소가 있고, 이중 3/4는 60.75이므로 60개의 등반 장소를 등반했을 경우 성공한 것으로 본다.

암벽등반 (계속)

입력

첫 번째 줄에는 농장의 크기 N이 주어진다.
다음 줄부터 N줄에 걸쳐서 N개의 영역별 높이가 공백으로 구분되어 주어진다.

N의 값은 500이하의 자연수이고, 각 지점의 돌출 정도는 0부터 1,000,000까지의 값 중 하나의 값을 가진다.

출력

등반을 성공적으로 마칠 수 있는 최소 H값을 출력한다.

입력 예	출력 예
5 9 9 9 3 3 0 0 0 0 3 0 9 9 3 3 0 0 0 3 3 9 0 0 9 3	3

풀이

이 문제는 지금까지 배웠던 알고리즘 설계법으로는 접근할 방법이 쉽게 떠오르지 않을 것이다. 따라서 이러한 문제들은 결정 문제로 바꾸고 이를 이분탐색으로 해결할 경우 의외로 쉽게 해결할 수 있는 경우가 많다.

먼저 주어진 입력 예시를 통하여 문제를 이해해보자. 주어진 암벽 지도는 다음과 같다.

9	9	9	3	3
0	0	0	0	3
0	9	9	3	3
0	0	0	3	3
9	0	0	9	3

만약 위의 지도에서 (1, 1)에서 출발한다고 가정하고 H를 10라고 가정하면 다음과 같은 영역을 모두 정복할 수 있다.

9	**9**	**9**	**3**	**3**
0	**0**	**0**	**0**	**3**
0	**9**	**9**	**3**	**3**
0	**0**	**0**	**3**	**3**
9	**0**	**0**	**9**	**3**

위와 같이 모든 암벽을 등반할 수 있다. 체력을 조금 더 줄여서 체력을 5라고 가정한 상태에서 (1, 1)에서 출발한다고 가정하면 다음과 같은 영역을 정복할 수 있다.

9	**9**	**9**	3	3
0	0	0	0	3
0	9	9	3	3
0	0	0	3	3
9	0	0	9	3

위 그림과 같이 회색 영역에서 다른 영역으로 이동하기 위해서는 최소 6이상의 체력이

필요하므로 더 이상의 이동은 불가능하다. 하지만 H가 5인 상태에서 (2, 1)에서 출발하면 다음과 같은 영역을 정복할 수 있다.

9	9	9	3	3
0	0	0	0	3
0	9	9	3	3
0	0	0	3	3
9	0	0	9	3

위의 경우 모두 18개의 영역을 정복했고, 전체 영역의 수는 25이므로 $25 \times \frac{3}{4} = 18.75$이므로 체력이 5인 경우는 가능하다. 사실 잘 살펴보면 체력이 3이어도 이 영역은 모두 정복 가능하다는 사실을 알 수 있다. 하지만 체력이 2이면 위 영역을 모두 정복할 수 없다. 따라서 이 경우는 체력이 최소 3이어야지만 등반을 성공할 수 있다.

이 문제를 해결하기 위해서 이분탐색법을 적용해보자. 먼저 해가 될 수 있는 값은 전체 맵에서 돌출 정도가 가장 큰 값을 넘을 수는 없다. 따라서 처음에 ub와 lb를 각각 0과 전체 max 값으로 설정하고 탐색을 시작할 수 있다.

먼저 다음과 같은 결정 문제를 해결하는 알고리즘을 작성하자.

체력 H로 전체 영역의 $\frac{3}{4}$을 정복할 수 있는가?

만약 위 결정 문제의 결과가 참이냐 거짓이냐에 따라서 다음의 절차에 따라 나머지 영역을 탐색하는 방법으로 처리하면 해를 구할 수 있다.

① lb와 ub를 1과 전체의 max값으로 설정한다.
② lb<ub일 동안 반복한다. 이 조건이 거짓이면 ⑤로 이동
③ m=(lb+ub-1)/2로 설정한다.
④ 만약 m으로 등반을 완료할 수 있다면 ub=m으로 설정하고 ②로 이동, 아니면 lb=m+1로 설정하고 계속 탐색
⑤ ub를 출력한다.

위의 결정 문제를 해결하기 위해서는 상, 하, 좌, 우로 값의 차가 H이하인 곳을 모두 탐색한다. 이 때 탐색은 깊이우선탐색, 너비우선탐색 모두 가능하다.

만약, 그 크기가 $\frac{3}{4}$ 이라면 아직 탐색하지 않은 영역을 새로운 출발지점으로 지정하고 다시 탐색을 시작한다. 또한, 모두 탐색을 완료했는데 최대 영역의 크기가 $\frac{3}{4}$ 미만이라면 H로는 탐색이 불가능하다고 판단할 수 있다.

이와 같은 방법으로 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <cstdio>
#include <cstring>

int N, M[511][511], chk[511][511];
int dx[4]={0,1,0,-1}, dy[4]={1,0,-1,0}, lb, ub, m;

int max(int a, int b) { return a>b ? a:b;}

int abs(int a) { return a>0 ? a:-a;}

bool can(int a, int b) { return (0<=a&&a<N)&&(0<=b&&b<N);}

int dfs(int a, int b, int h)
{
  int area=1;
  chk[a][b]=true;
  for(int i=0; i<4; i++)
    if(can(a+dx[i],b+dy[i]) && !chk[a+dx[i]][b+dy[i]] &&
      abs(M[a][b]-M[a+dx[i]][b+dy[i]])<=h)
      area+=dfs(a+dx[i],b+dy[i],h);
  return area;
}

bool f(int h)
{
  int mcnt=0, cnt;
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
      if(!chk[i][j])
      {
        cnt=dfs(i,j,h);
        mcnt=max(cnt, mcnt);
```

코드	참고

```
      }
  return (mcnt>=(int)(N*N*0.75));
}

int main()
{
  scanf("%d",&N);
  for(int i=0; i<N; i++)
    for(int j=0; j<N; j++)
    {
      scanf("%d", &M[i][j]);
      ub=max(ub, M[i][j]);
    }
  while(lb<ub)
  {
    memset(chk, false, sizeof(chk));
    m=(lb+ub-1)/2;
    if(f(m)) ub=m;
    else lb=m+1;
  }
  printf("%d\n", ub);
  return 0;
}
```

이 소스코드에는 지금까지 배웠던 다양한 내용들을 많이 포함하고 있으므로, 잘 익혀두는 편이 좋다.

만약 이 문제에서 속도를 조금 더 빠르게 하고자 한다면 시작 위치를 찾기 위해 N^2개의 모든 영역을 탐색하는데, 실제로는 전체 영역의 $\frac{1}{4}$보다만 더 크게 탐색하면 된다. 왜냐하면 적어도 전체의 $\frac{3}{4}$이상을 정복해야 성공할 수 있으므로 $\frac{1}{4}$보다 1이상 큰 영역에 대해서 반드시 정복영역에 포함된 영역이 존재할 수밖에 없다. 따라서 다음과 같이 수정하면 경우에 따라서 효율이 더 좋아질 수 있다.

```
bool f(int h)
{
  int mcnt=0, cnt;
  for(int i=0; i<N/2+1; i++)
    for(int j=0; j<N/2+1; j++)
      if(!chk[i][j])
      {
        cnt=dfs(i,j,h);
        mcnt=max(cnt, mcnt);
      }
  return (mcnt>=(int)(N*N*0.75));
}
```

문제 3

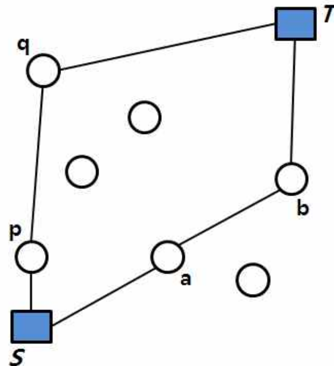
경비행기(L)

경비행기 독수리호가 출발지 S에서 목적지 T로 가능한 빠른 속도로 안전하게 이동하고자 한다. 이 때, 경비행기의 연료통의 크기를 정하는 것이 중요한 문제가 된다.

큰 연료통을 장착하면 중간에 내려서 급유를 받는 횟수가 적은 장점이 있지만 연료통의 무게로 인하여 속도가 느려지고, 안정성에도 문제가 있을 수 있다.

한편 작은 연료통을 장착하면 비행기의 속도가 빨라지는 장점이 있지만 중간에 내려서 급유를 받아야 하는 횟수가 많아지는 단점이 있다.

문제는 중간에 내려서 급유를 받는 횟수가 K 이하일 때 연료통의 최소 용량을 구하는 것이다. 아래 예를 보자.



위 그림은 S, T와 7개의 중간 비행장의 위치를 나타내고 있는 그림이다. 위 예제에서 중간급유를 위한 착륙 허용 최대횟수 K = 2라면, S-a-b-T로 가는 항로가 S-p-q-T로 가는 항로보다 연료통이 작게 된다.

왜냐하면, S-p-q-T 항로에서 q-T의 길이가 매우 길어서 이 구간을 위해서는 상당히 큰 연료통이 필요하기 때문이다.

문제는 이와 같이 중간에 최대 K번 내려서 갈 수 있을 때 최소 연료통의 크기가 얼마인지를 결정하여 출력하는 것이다.

참고사항은 다음과 같다.

경비행기(L) (계속)

1) 모든 비행기는 두 지점 사이를 반드시 직선으로 날아간다. 거리의 단위는 km 이며 연료의 단위는 ℓ(리터)이다. 1ℓ당 비행거리는 10km이고 연료 주입은 ℓ단위로 한다.

2) 두 위치간의 거리는 평면상의 거리이다. 예를 들면, 두 점 g = (2, 1)와 h = (37, 43) 간의 거리 d(g, h)는 sqrt{ (2-37)^2 + (1-43)^2 }= 54.671이고 50 < d(g, h) ≤ 60이므로 필요한 연료는 6ℓ가 된다.

3) 출발지 S의 좌표는 항상 (0, 0) 이고 목적지 T의 좌표는 (10000,10000) 으로 모든 입력 데이터에서 고정되어 있다.

4) 출발지와 목적지를 제외한 비행장의 수 N은 3 <= N <= 10이고 그 좌표값 (x, y)의 범위는 0 < x, y < 10,000인 정수이다. 그리고 0 <= K <= 1000이다.

입력

입력의 첫 줄에는 N 과 K 가 하나의 공백을 두고 주어진다. 그 다음 N개의 줄에는 각 비행장(급유지)의 정수좌표가 "x y"의 형식으로 주어진다.

출력

출력에는 S에서 T까지 K번 이하로 중간급유를 하여 갈 수 있는 항로에서의 최소 연료통 용량에 해당하는 정수를 출력한다.

입력 예	출력 예
10 1 10 1000 20 1000 30 1000 40 1000 5000 5000 1000 60 1000 70 1000 80 1000 90 7000 7000	708

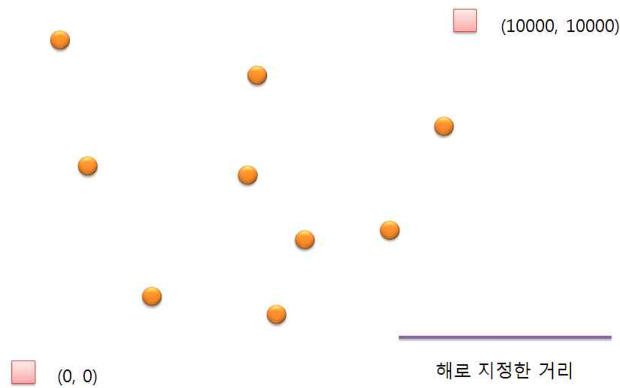
출처: 한국정보올림피아드(2005 전국본선 고등부)

풀이

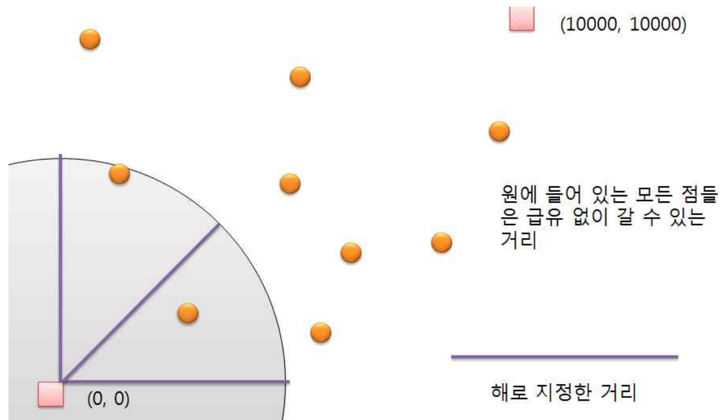
이 문제도 특별한 아이디어가 잘 떠오르지 않는 문제이다. 따라서 이 문제도 결정 문제를 해결한 후 이 결과를 이분탐색을 이용하여 해결해보자. 먼저 결정 문제를 다음과 같이 정의할 수 있다.

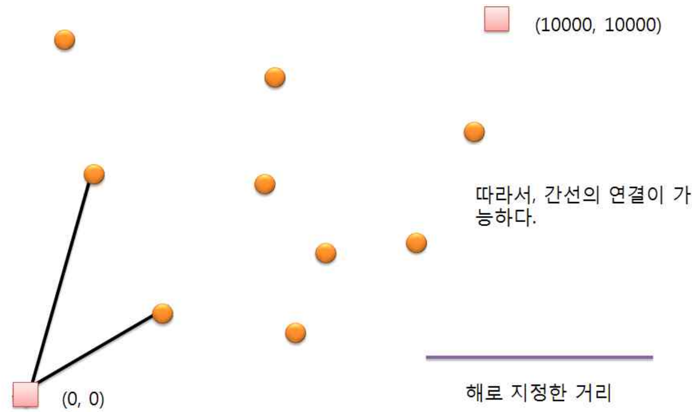
> L리터의 연료통으로 k개 이하의 경유지를 거쳐 목적지에 도달할 수 있는가?

이 문제를 해결하는 다양한 방법이 있지만 기본적으로 큐를 이용한 너비우선 탐색법으로 해결할 수 있다. 먼저 다음 예를 통해서 알아보자.

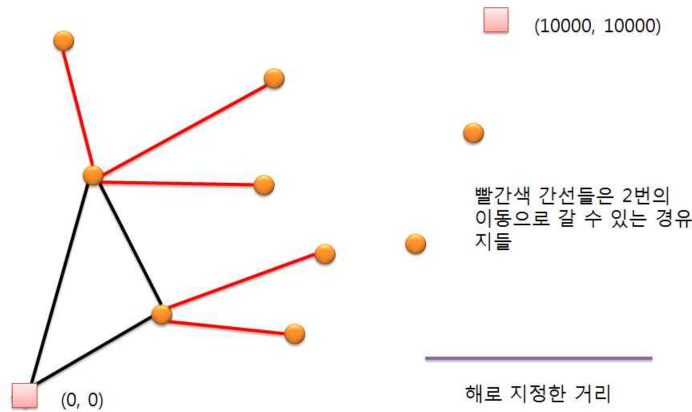


위의 그림에서 (0, 0)은 출발점이고 (10000, 10000)은 도착점을 의미하며, 각 점은 급유 가능한 경유지를 의미한다. 그리고 먼저 L리터로 갈 수 있는 거리를 지정한 후, 출발지로부터 지정한 거리 이내에 있는 모든 경유지를 큐에 삽입한다.





그림과 같이 처음으로 들를 수 있는 경유지는 다음과 같이 2개이며, 이 값을 큐에 삽입하고 계속 탐색을 진행한다.



다음으로 보기와 같이 5개의 경유지에 도달할 수 있다. 하지만 이미 k의 값은 1이므로 더 이상의 경유지에 들를 수 없다. 따라서 위 5개의 위치 중 도착점이 있으면 L리터로 이동 가능한 것이다. 하지만 위의 예에서는 2번에 이동 가능한 5개의 영역 중 도착점이 존재하지 않으므로 이동 불가능한 경우가 된다.

따라서 위와 같은 경우는 L을 더 늘려서 너비우선탐색을 다시 진행하는 과정을 거쳐야 한다. 따라서 알고리즘은 다음과 같은 단계로 작성할 수 있다.

① lb와 ub를 0과 전체의 14142로 설정한다.
② lb 〈 ub일 동안 반복한다. 이 조건이 거짓이면 ⑤로 이동
③ m = (lb + ub - 1)/2로 설정한다.
④ 만약 m으로 목적지까지 이동 가능하다면 ub = m으로 설정하고 ②로 이동, 아니면 lb = m+1로 설정하고 계속 탐색
⑤ ub를 출력한다.

위의 원리를 이용하여 작성한 소스코드는 다음과 같다.

```
#include <cstdio>
#include <cmath>
#include <queue>
#include <cstring>

using namespace std;

struct P{int x, y;};
int N, K, T, chk[1013], D[1013][1013];
P S[1013];

int f(int t)
{
  queue<int> Q;
  memset(chk,-1, sizeof(chk));
  Q.push(0), chk[0]=0;
  while(!Q.empty())
  {
    int i;
    T=Q.front(), Q.pop();
    if(chk[T]>K+1) return 0;
    else if(T==N+1) return 1;
    for(i=0; i<=N+1; i++)
    {
      if(chk[i]==-1 && D[T][i]<t*10)
      {
        chk[i]=chk[T]+1;
        Q.push(i);
      }
```

코드	참고

```
    }
  }
  return 0;
}

int main()
{
  int i, j, lb=0, ub=14142, m, x, y;
  scanf("%d %d", &N, &K);
  S[0].x=S[0].y=0,S[N+1].x=S[N+1].y=10000;
  for(i=1; i<=N; i++) scanf("%d%d",&S[i].x,&S[i].y);
  for(i=0; i<=N+1; i++)
    for(j=0; j<=N+1; j++)
      D[i][j]=(int)sqrt((S[i].x-S[j].x)*(S[i].x-S[j].x)
             +(S[i].y-S[j].y)*(S[i].y-S[j].y));
  while(lb<ub)
  {
    m=(ub+lb-1)/2;
    if(f(m)) ub=m;
    else lb = m+1;
  }
  printf("%d\n", ub);
  return 0;
}
```

5 자료구조를 활용한 알고리즘의 고속화

이 단원에서는 보다 효율적인 자료처리를 위한 자료구조의 활용에 대해서 간단히 다룬다. 물론 다양한 형태의 자료구조가 존재하지만 일반적으로 가장 활용도가 높은 자료구조인 index tree로 불리는 complete binary segment tree에 대해서 소개한다.

먼저 이 자료구조를 알기 전에 정해진 구간 내에서 구간합을 구하거나 구간의 최댓값, 최솟값 등을 구해야 하는 문제들이 등장할 때에 활용할 수 있는 방법을 예제를 통해서 알아보자.

예

임의의 정수 n을 입력받은 후, q개의 쿼리에 대해서 답을 출력하는 프로그램을 작성하시오.

단, 쿼리는 두 개의 정수 a, b로 구성되며, 쿼리에 대한 답은 구간 a부터 b까지의 합을 출력하면 된다. 단 구간의 합은 a와 b를 모두 포함하는 범위의 합을 출력해야 한다.
(단, n, $q \le 100,000$; $1 \le a \le b \le n$; 각 원소의 크기는 1,000 이하의 자연수)

풀이

이 문제는 매우 간단한 문제처럼 보이지만 사실 10만개의 원소에 대해서 10만 번이나 입력되는 쿼리를 처리하기 위해서, 문제에서 주어진 대로 구간을 입력받고, 각 구간의 합을 일반적인 방법으로 구한다면, 계산량이 nq가 되어 엄청난 시간이 걸리게 된다.

일반적으로 해결하는 소스코드는 다음과 같다.

```
#include <cstdio>

int S[100001], n, q;

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%d", S+i);
  scanf("%d", &q);
  while(q--)
  {
    int a,b,sum=0;
    scanf("%d %d", &a, &b);
    for(int i=a; i<=b; i++)
      sum+=S[i];
    printf("%d\n", sum);
  }
  return 0;
}
```

위와 같이 작성한 알고리즘은 너무 많은 시간이 걸려서 제한 시간 내에 답을 출력할 수가 없다. 따라서 어떤 조치를 취하지 않으면 안 된다.

이 문제의 경우는 동적표를 이용하여 1부터 k까지의 합을 미리 구해두어 이를 이용하여 구간의 합을 쉽게 구할 수 있다. 다음 예를 통하여 알아보자.

다음과 같이 10 개의 데이터가 순서대로 입력되었다고 생각하자.

```
10
9 8 1 7 15 2 4 19 3 6
```

이때, 누적합 배열을 간단히 생각해 보면 아래와 같다.

입력된 배열 A

9	8	1	7	15	2	4	19	3	6
0	1	2	3	4	5	6	7	8	9

누적합 배열 D

9	17	18	25	40	42	46	65	68	74
0	1	2	3	4	5	6	7	8	9

위와 같이 누적합 배열을 계산량 n으로 구성해두면 다음부터 쿼리에 대한 처리를 한 번의 연산으로 바로 구현할 수 있다. 예를 들어 a와 b의 구간의 합을 계산하려면 다음과 같이 하면 된다.

$$D[b] - D[a-1]$$

$D[b]$는 1부터 b까지의 합을 가지고 있고 $D[a-1]$은 1부터 $a-1$까지의 합을 가지고 있으므로 두 값의 차는 a부터 b까지의 합이 되므로 상수시간에 계산이 가능하다. 이 방법을 활용하여 구현한 소스코드는 다음과 같다.

```
#include <cstdio>

int S[100001],D[100001],n,q;

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%d", S+i);
  for(int i=0; i<n; i++)
    if(i==0) D[i]=S[i];
    else D[i]=D[i-1]+S[i];
  scanf("%d", &q);
  while(q--)
  {
    int a, b;
    scanf("%d %d", &a, &b);
    if(a==0)
      printf("%d\n", D[b]);
    else
      printf("%d\n", D[b]-D[a-1]);
  }
  return 0;
}
```

이와 같이 누적합을 이용하면 구간합은 쉽게 해결할 수 있다. 다음으로 조금 더 진보된 예제를 해결해보자.

예

임의의 정수 n을 입력받은 후, q개의 2가지 쿼리에 대해서 답을 출력하는 프로그램을 작성하시오.

첫 번째 쿼리는 처음에 1이 입력되고 다음으로 두 개의 정수 a, b로 구성되며, 쿼리에 대한 답은 구간 a부터 b까지의 합을 출력하면 된다.

두 번째 쿼리는 처음에 2가 입력되고 다음으로 두 개의 정수 a, b가 입력된다. 이 쿼리는 a번째로 저장된 값을 b로 수정해야 한다.

(단, n, $q \leq 100{,}000$; $1 \leq a \leq b \leq n$; 각 원소의 크기는 1,000 이하의 자연수)

풀이

이번 문제는 앞의 문제와 달리 가지고 있던 값을 바꾸는 쿼리가 존재한다. 즉 값을 갱신해야 하므로, 누적합을 구하는 방법으로 처리할 수 없다. 왜냐하면 2번 쿼리가 발생될 때마다 누적합을 갱신해야 하기 때문이다.

누적합을 갱신하려면 다시 n만큼의 계산량이 들기 때문에 누적합을 만들어 두어서 얻을 수 있는 장점을 모두 잃게 된다. 따라서 또 다른 방법이 필요하다.

이러한 상황을 대처할 수 있는 여러 가지 자료구조가 존재한다. 먼저 버킷을 이용한 방법이 있고, 다음으로 BIT(binary index tree)라는 자료구조가 존재한다. 특히 BIT는 매우 강력한 방법이긴 하지만 일반적으로 누적합만 활용할 수 있으므로, 이 교재에서는 보다 일반적으로 활용될 수 있는 index tree로 해결하는 방법을 소개한다. 여기서 소개하는 index tree는 complete binary tree로 구성된 segment tree를 말하며 앞으로 IDT라고 부른다.

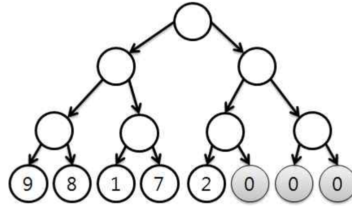
IDT는 정해진 구간의 합, 최댓값, 최솟값 등을 lg n의 시간에 구할 수 있으며, 갱신 또한 lg n시간에 처리할 수 있는 매우 효율적인 자료구조이다. 기본적인 구조는 원본 데이터를 complete binary tree의 단말노드에 배치하고, 그 값의 부모노드를 특정한 의미를 가지는 index로 활용하는 방법이다.

만약 구간합을 계산하는 IDT를 만들고자 한다면 다음과 같이 구성한다. 5개의 데이터를 가지고 IDT를 구성한 그림은 다음과 같다.

```
5
9 8 1 7 2
```

일반적으로 IDT는 complete binary tree를 가장 효율적으로 표현할 수 있는 일차원 배열을 이용하고, 그 배열의 크기는 원본 data가 5개 이므로 5이상인 가장 작은 2^k값을 기준으로 설정한다.

여기서는 8이 되므로 이 값의 2배에서 1을 뺀 15개의 원소를 가지는 배열로 구성할 수 있다. 트리의 모양을 나타내면 다음과 같다.



초기의 값을 입력받은 후의 IDT의 값

								9	8	1	7	2	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

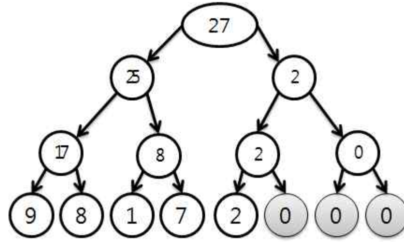
IDT를 처음 설계하고 처음 입력받는 값을 base라고 한다. base는 입력크기와 같거나 큰 가장 작은 2^k값으로 한다.

여기까지를 구현한 코드는 다음과 같다.

줄	코드	참고
1	`#include <cstdio>`	
2		
3	`int IDT[1<<18], n, base;`	
4		
5	`int main()`	
6	`{`	
7	`  scanf("%d", &n);`	
8	`  for(base=1; base<n; base*=2);`	
9	`  for(int i=base; i<n+base; i++)`	
10	`    scanf("%d", &IDT[i]);`	
11		
12	`  return 0;`	
13	`}`	

여기서 8행은 base를 설정하기 위한 부분이다. 눈 여겨 볼 부분은 데이터를 0번부터 입력받는 것이 아니라 base부터 받는 것이다. 이는 단말노드에 값들을 배치하기 위함이다.

다음으로 index를 만들어 보자. 이 문제에서는 부분합이 필요하므로, 부모는 두 자식의 값의 합으로 채워 올리면 된다.



인덱스 값을 채운 후의 IDT의 값

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	27	25	2	17	8	2	0	9	8	1	7	2	0	0	0

회색의 부분이 인덱스로 만들어진 부분이다. 이와 같이 1번을 루트로 하는 완전이진트리가 완성되고 단말노드의 값들이 원래 입력받은 데이터가 된다.

인덱스를 구성하는 코드는 다음과 같다.

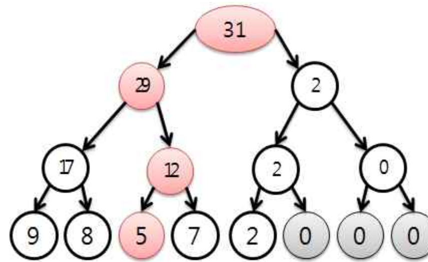
```
#include <cstdio>

int IDT[1<<18], n, base;

int main()
{
  scanf("%d", &n);
  for(base=1; base<n; base*=2);
  for(int i=base; i<n+base; i++)
    scanf("%d", &IDT[i]);
  for(int i=base-1; i>0 ; i--)
    IDT[i]=IDT[2*i]+IDT[2*i+1];
  return 0;
}
```

이와 같이 간단하게 인덱스를 모두 구성할 수 있다. 다음으로 가장 중요한 연산인 구간 합을 구하는 것과 값을 갱신하는 방법에 대해서 알아보자.

먼저 값의 갱신은 다음과 같이 이루어진다. 만약 3번째 원소를 5로 바꾸면 다음과 같은 과정을 거쳐서 lg n만에 갱신된다.



-	31	29	2	17	12	2	0	9	8	5	7	2	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

이와 같이 매우 효율적으로 전체를 갱신할 수 있다. 이 갱신하는 코드는 다음과 같이 구현할 수 있다.

코드	참고

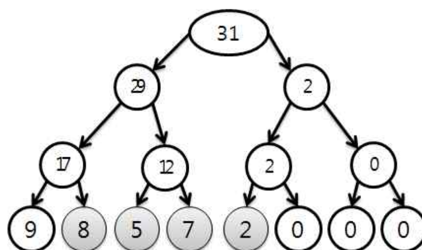
```
#include <cstdio>

int IDT[1<<18], n, base;

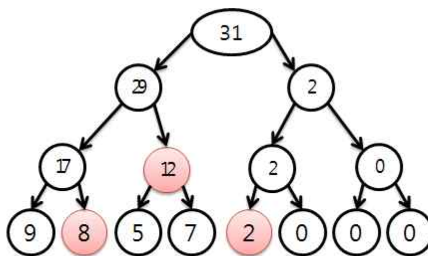
void update(int a, int b)
{
  IDT[a]=b;
  a>>=1;
  while(a)
  {
    IDT[a]=IDT[2*a]+IDT[2*a+1];
    a>>=1;
  }
}

int main()
{
  scanf("%d", &n);
  for(base=1; base<n; base*=2);
  for(int i=base; i<n+base; i++)
    scanf("%d", &IDT[i]);
  for(int i=base-1; i>0; i--)
    IDT[i]=IDT[2*i]+IDT[2*i+1];
  return 0;
}
```

마지막으로 구간의 합을 구해보자. 구간의 시작을 a 구간의 끝을 b라고 할 때, a, b의 값이 짝수인지 홀수인지에 따라서 구현하는 방법이 달라진다. 다음 그림을 통하여 구간 2부터 5까지의 합을 구해보자.



위의 그림과 같이 구간이 설정된다.



그림과 같이 3개의 값만 더하면 된다. 구간이 아무리 넓더라도 루트까지 올라가면 모든 구간의 합을 구할 수 있으므로 lg n에 모든 합을 처리할 수 있다.

여기서 중요한 것은 배열의 인덱스 값을 기준으로 구간의 시작은 짝수, 구간의 끝은 홀수이면 그 구간의 합을 구하기 위해서 부모노드를 참조할 수 있지만, 구간의 시작이 홀수라면 부모노드가 다른 값을 포함하고 있으므로, 현재 홀수 값을 따로 합한 다음 구간을 [a+1, b]로 변경하여 다시 합을 구한다.

마찬가지로 구간의 끝이 짝수라면 구간을 [a, b−1]로 재설정하여 다시 합을 구하는 과정을 반복한다.

위 그림에서 8과 2는 부모로 갈 수 없는 값들이라 따로 합을 저장하고 구간이 [a+1, b−1]로 변경되어 합을 구하는 과정을 보여준다. 이 과정을 소스코드로 구현한 결과는 다음과 같다.

```
#include <cstdio>

int IDT[1<<18], n, base;

int lg_sum(int a, int b)
{
  int sum=0;
  while(a<b)
  {
    if(a%2==1) sum+=IDT[a],a++;
    if(b%2==0) sum+=IDT[b],b--;
    a>>=1, b>>=1;
  }
  if(a==b) sum+=IDT[a];
  return sum;
}

void update(int a, int b)
{
  IDT[a]=b;
  a>>=1;
  while(a)
  {
    IDT[a]=IDT[2*a]+IDT[2*a+1];
    a>>=1;
  }
}

int main()
{
  scanf("%d", &n);
  for(base=1; base<n; base*=2);
  for(int i=base; i<n+base; i++)
    scanf("%d", &IDT[i]);
  for(int i=base-1; i>0; i--)
    IDT[i]=IDT[2*i]+IDT[2*i+1];
  return 0;
}
```

이 IDT구조를 이용하여 예제를 해결한 소스코드는 다음과 같다. 여기서는 main()만 표현하고 나머지 함수들은 위 소스코드를 참고하기 바란다.

```
#include <cstdio>

int main()
{
  scanf("%d", &n);
  for(base=1; base<n; base*=2);
  for(int i=base; i<n+base; i++)
    scanf("%d", &IDT[i]);
  for(int i=base-1; i>0; i--)
    IDT[i]=IDT[2*i]+IDT[2*i+1];
  scanf("%d", &q);
  while(q--)
  {
    int c,a,b;
    scanf("%d %d %d", &c, &a, &b);
    if(c==1)
    {
      printf("%d\n", lg_sum(base+a-1,base+b-1));
    }
    else
    {
      update(base+a-1, b);
    }
  }
  return 0;
}
```

이와 같이 처리하면 lg n에 모든 쿼리를 처리할 수 있으므로 매우 효율적인 알고리즘이 된다. 따라서 전체 계산량은 $q$lg n이 된다.

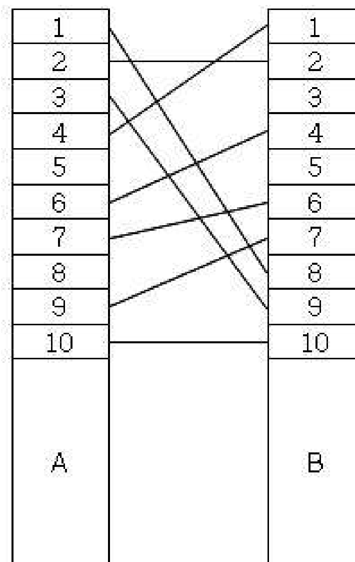
이제 실제 문제들을 통하여 자료구조를 활용한 기법을 익힐 수 있도록 해보자.

문제 1

전깃줄(L)

두 전봇대 A와 B 사이에 하나 둘씩 전깃줄을 추가하다 보니 전깃줄이 서로 교차하는 경우가 발생하였다. 합선의 위험이 있어 이들 중 몇 개의 전깃줄을 없애 전깃줄이 교차하지 않도록 만들려고 한다.

예를 들어, <그림 1>과 같이 전깃줄이 연결되어 있는 경우 A의 1번 위치와 B의 8번 위치를 잇는 전깃줄, A의 3번 위치와 B의 9번 위치를 잇는 전깃줄, A의 4번 위치와 B의 1번 위치를 잇는 전깃줄을 없애면 남아있는 모든 전깃줄이 서로 교차하지 않게 된다.



〈그림 1〉

전깃줄이 전봇대에 연결되는 위치는 전봇대 위에서부터 차례대로 번호가 매겨진다. 전깃줄의 개수와 전깃줄들이 두 전봇대에 연결되는 위치의 번호가 주어질 때, 남아있는 모든 전깃줄이 서로 교차하지 않게 하기 위해 없애야 하는 전깃줄의 최소 개수를 구하는 프로그램을 작성하시오.

전깃줄(L) (계속)

입력

첫째 줄에는 두 전봇대 사이의 전깃줄의 개수가 주어진다. 전깃줄의 개수는 100이하의 자연수이다.

둘째 줄부터 한 줄에 하나씩 전깃줄이 A전봇대와 연결되는 위치의 번호와 B전봇대와 연결되는 위치의 번호가 차례로 주어진다.

위치의 번호는 500 이하의 자연수이고, 같은 위치에 두 개 이상의 전깃줄이 연결될 수 없다.

출력

첫째 줄에 남아있는 모든 전깃줄이 서로 교차하지 않게 하기 위해 없애야 하는 전깃줄의 최소 개수를 출력한다.

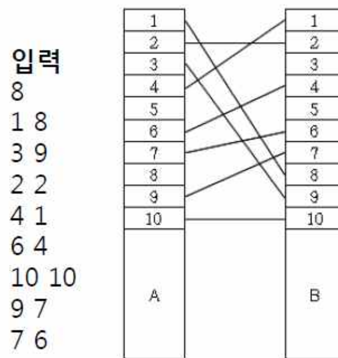
입력 예	출력 예
8 1 8 3 9 2 2 4 1 6 4 10 10 9 7 7 6	3

출처: 한국정보올림피아드(2007 지역본선 중고등부)

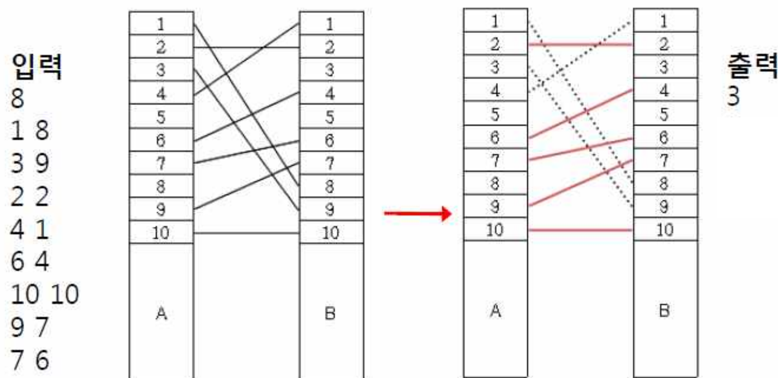
풀이

두 전봇대 A, B에 연결된 전깃줄이 서로 교차하지 않도록 전깃줄을 제거하는 문제이다. 단, 최소의 전깃줄을 제거해야 한다. 전깃줄의 개수는 100,000이하이며 위치번호는 500,000 이하 자연수이다. 또한 같은 위치에 전깃줄은 없다.

입력이 왼쪽과 같다면 아래 그림과 같은 상태를 표현한 것이다.



A 전봇대에서 1, 3, 4번 전깃줄을 제거하면 나머지 모든 전깃줄이 서로 교차하지 않게 되고, 최소 제거 개수는 3이 된다.



만약, "가장 많이 교차하는 전깃줄을 제거해 가면서 교차여부를 판단해보면 쉽게 해결할 수 있을 것이다"라는 가정을 한다면, A 기둥에 연결되어있는 각 전깃줄에 교차횟수를 계산하고 기록한 후, 가장 많은 교차 횟수를 가진 것부터 제거해 가면서 교차 여부를 확인하는 방법을 사용할 수 있다.

교차수	A	B
5	1	1
2	2	2
4	3	3
3	4	4
0	5	5
2	6	6
2	7	7
0	8	8
2	9	9
0	10	10

#1

교차수 최대인 전깃줄 1 제거

교차수	A	B
0	1	1
1	2	2
4	3	3
2	4	4
0	5	5
1	6	6
1	7	7
0	8	8
1	9	9
0	10	10

제거 후 교차수 재계산

교차수	A	B
0	1	1
1	2	2
4	3	3
2	4	4
0	5	5
1	6	6
1	7	7
0	8	8
1	9	9
0	10	10

#2

교차수 최대인 전깃줄 4 제거

교차수	A	B
0	1	1
1	2	2
0	3	3
1	4	4
0	5	5
0	6	6
0	7	7
0	8	8
0	9	9
0	10	10

제거 후 교차수 재계산

교차수	A	B
0	1	1
0	2	2
0	3	3
0	4	4
0	5	5
0	6	6
0	7	7
1	8	8
0	9	9
1	10	10

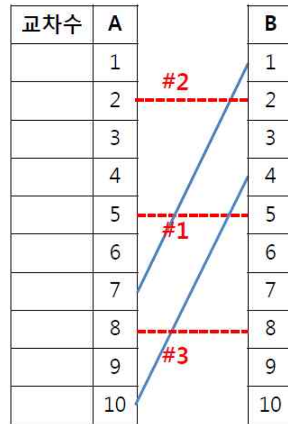
#3

교차수 최대인 전깃줄 8 제거

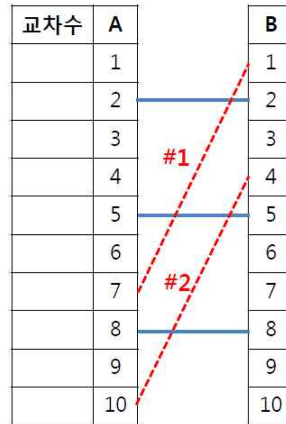
교차수	A	B
0	1	1
0	2	2
0	3	3
0	4	4
0	5	5
0	6	6
0	7	7
0	8	8
0	9	9
0	10	10

제거 후 교차수 재계산

3개를 제거하면 2개가 남지만, 2개를 제거하고 3개를 남기는 방법도 있다. 따라서 교차수가 가장 많은 전깃줄을 순서대로 제거해 나가는 방식은 적당하지 않다.

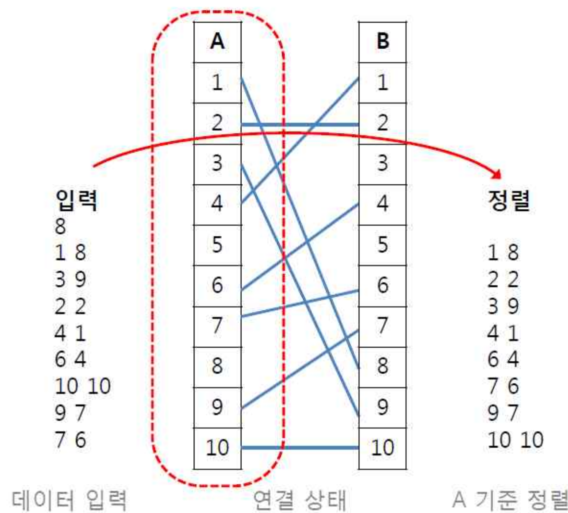


3개 제거 방법?



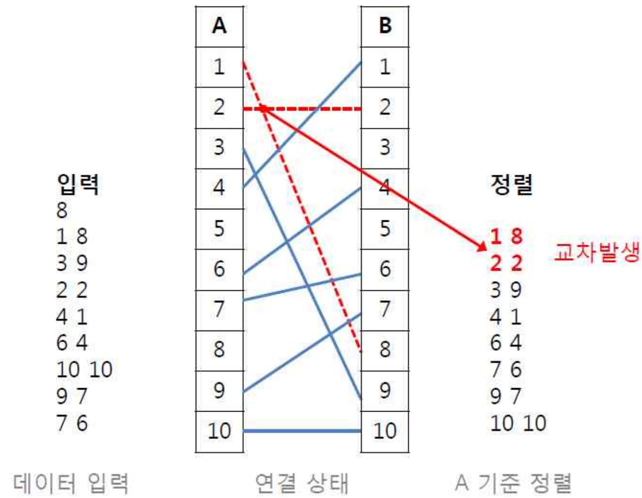
하지만! 2개만 제거해도 된다!!

두 전봇대의 전기줄의 연결 순서를 생각하고, 한 전봇대를 기준으로 연결관계를 생각해 보면, 보다 간단하고 정확한 문제해결 전략을 만들어낼 수 있다.

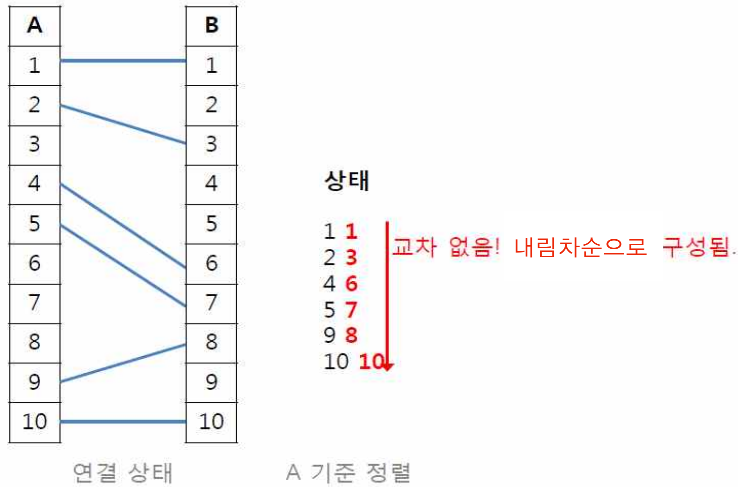


데이터 입력 연결 상태 A 기준 정렬

A를 기준으로 두 쌍의 데이터를 정렬하고 관찰해 보면, B 에서 위에서 아래로 오름차순이 아닌 경우 두 전깃줄이 서로 교차됨을 확인할 수 있다.



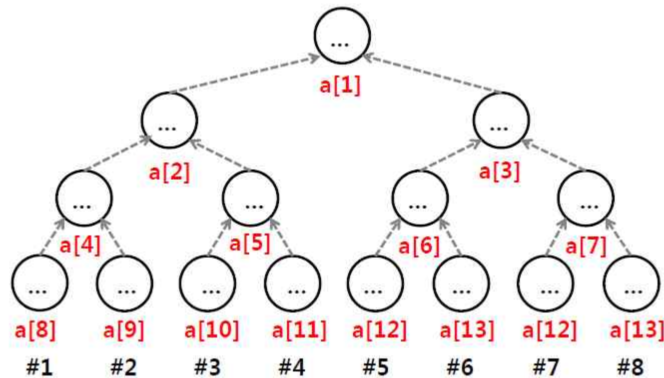
따라서 만약, A에 연결되어있는 순서가 오름차순이고, B에 연결되어있는 순서도 모두 오름 차순 이라면? 두 기둥에 연결되어있는 전깃줄은 서로 교차하지 않게 된다는 것을 의미한다.



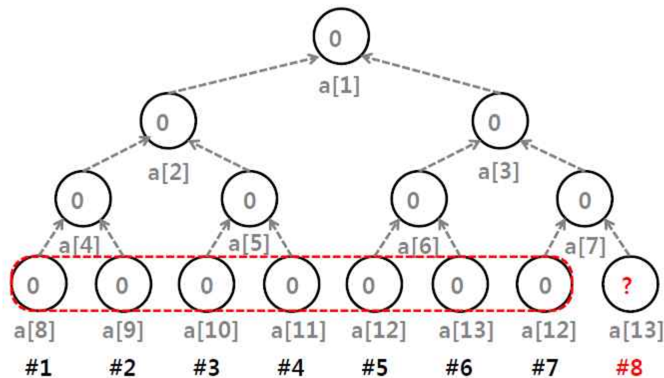
따라서, 입력된 데이터를 A를 기준으로 정렬 시킨 후, B의 데이터로 만들 수 있는 가장 긴 오름 수열을 찾아내면, 그 수열의 길이가 바로 서로 교차하지 않는 전깃줄의 최대 개수가 되는 것이다.

이 문제는 이분탐색을 통해서도 해결할 수 있지만 이 단원에서는 IDT를 이용한 풀이를 소개한다.

이 문제의 핵심은 효과적인 문제해결전략의 핵심인 LIS 를 효과적으로 빠르게 찾아내는 것이다. 특히, 먼저 입력된 데이터들로 만들 수 있는 최대 LIS 길이를 찾는 것이 중요한데, IDT를 사용하는 인덱스 트리를 만들면 O(lg n) 만에 빠르게 찾아낼 수 있다.



8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자. 만약 8이 먼저 입력된다고 한다면 8보다 작은 수가 이전에 들어왔었는지를 IDT를 보고 검사할 수 있다.



이 영역의 구간의 합을 구한다. 이 값의 의미는 8이전에 입력된 데이터들 중 8미만의 데이터의 최댓값을 빠른 시간에 찾을 수 있다.



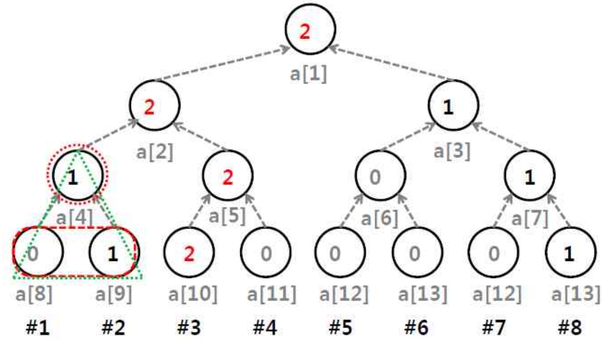
위에서 빨간 원으로 표시된 3개의 값만 비교하면 아래 구간의 최댓값을 찾을 수 있다. 따라서 최댓값이 0이므로 0+1의 값을 8번 자리에 기록한 후, 이를 갱신한다. 갱신하는 과정은 다음 그림과 같다.

이와 같이 $\lg n$ 시간에 루트노드까지 갱신할 수 있다. 이와 같이 입력되는 값들을 처리하는 과정은 다음과 같다.

8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.
2가 다음에 입력된다고 할 때...

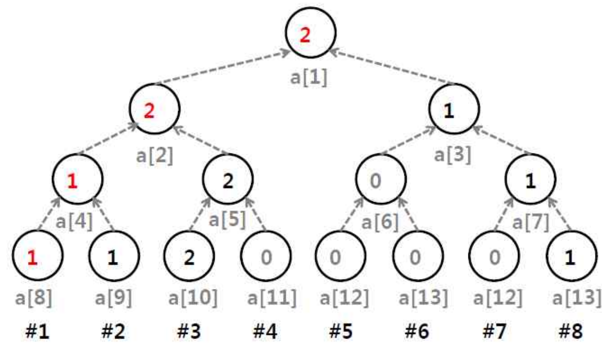
8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.

3이 다음에 입력된다고 할 때...



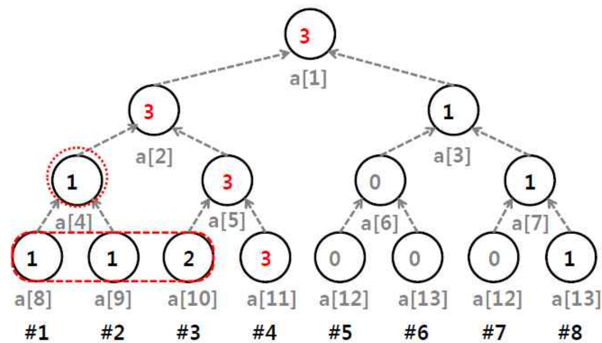
8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.

1이 다음에 입력된다고 할 때...

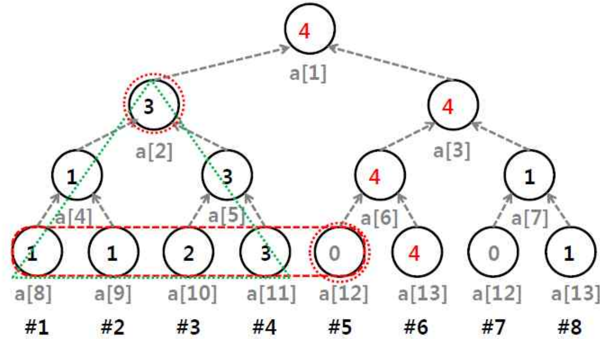


8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.

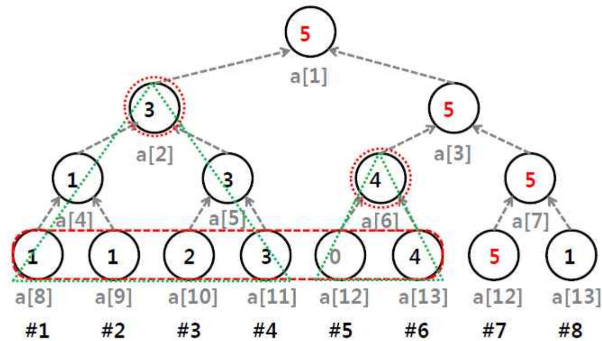
4가 다음에 입력된다고 할 때...



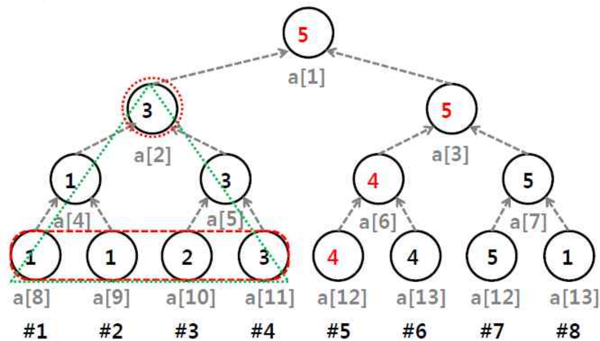
8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.
6이 다음에 입력된다고 할 때...



8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.
7이 다음에 입력된다고 할 때...



8 2 3 1 4 6 7 5 가 순서대로 입력된다고 하자.
5가 다음에 입력된다고 할 때...



이와 같은 단계를 거쳐서 $n\lg n$의 계산량으로 이와 같이 처리가 가능하다. 마지막에 출력할 값은 단말노드들 중에서 최댓값을 출력할 수 있다. 이와 같은 아이디어를 소스코드로 작성한 결과는 다음과 같다.

코드	참고

```
#include <stdio.h>
#include <algorithm>
using namespace std;
struct w {int a, b;} d[100001];
int IDT[1<<22];
bool cmp(w x, w y){ return x.a<y.a;}

void update(int n)
{
  while(n>1)
  {
    if(IDT[n]>IDT[n/2]) IDT[n/2]=IDT[n];
    n>>=1;
  }
}

int lg_sum(int s, int e)
{
  int m=0;
  while(s<e)
  {
    m=max(m, IDT[e]);
    if(!(e%2)) e--;
    e>>=1, s>>=1;
  }
  if(s==e) m=max(m, IDT[s]);
  return m;
}

int main()
{
  int i, base, n, m=-1;
  scanf("%d", &n);
  for(i=1; i<=n; i++)
  {
    scanf("%d %d", &d[i].a, &d[i].b);
```

```
    if(m<d[i].b) m=d[i].b;
  }
  sort(d+1,d+n+1,cmp);
  for(base=1; base<m; base<<=1);
  for(i=1; i<=n; i++)
  {
    IDT[base+d[i].b-1]=lg_sum(base,base+d[i].b-2)+1;
    update(base+d[i].b-1);
  }
  printf("%d\n", n-IDT[1]);
  return 0;
}
```

문제 2

달리기

KOI 장거리 달리기 대회가 진행되어 모든 선수가 반환점을 넘었다. 각 선수의 입장에서 자기보다 앞에 달리고 있는 선수들 중 평소 실력이 자기보다 좋은 선수를 남은 거리 동안 앞지르는 것은 불가능하다. 반대로, 평소 실력이 자기보다 좋지 않은 선수가 앞에 달리고 있으면 남은 거리 동안 앞지르는 것이 가능하다. 이러한 가정 하에서 각 선수는 자신이 앞으로 얻을 수 있는 최선의 등수를 알 수 있다.

각 선수의 평소 실력은 정수로 주어지는데 더 큰 값이 더 좋은 실력을 의미한다. 현재 달리고 있는 선수를 앞에서부터 표시했을 때 평소 실력이 각각 2, 8, 10, 7, 1, 9, 4, 15라고 하면 각 선수가 얻을 수 있는 최선의 등수는 (같은 순서로) 각각 1, 1, 1, 3, 5, 2, 5, 1이 된다.

예를 들어, 4번째로 달리고 있는 평소 실력이 7인 선수는 그 앞에서 달리고 있는 선수들 중 평소 실력이 2인 선수만 앞지르는 것이 가능하고 평소실력이 8과 10인 선수들은 앞지르는 것이 불가능하므로, 최선의 등수는 3등이 된다.

입력

첫째 줄에는 선수의 수를 의미하는 정수 n이 주어진다. n은 3 이상 500,000 이하이다.

이후 n개의 줄에는 정수가 한 줄에 하나씩 주어진다. 이 값들은 각 선수들의 평소 실력을 앞에서 달리고 있는 선수부터 제시한 것이다. 각 정수는 1 이상 1,000,000,000 이하이다. 단, 참가한 선수들의 평소 실력은 모두 다르다.

출력

각 선수의 최선의 등수를 나타내는 정수 개를 입력에 주어진 선수 순서와 동일한 순서로 한 줄에 하나씩 출력한다.

입력 예	출력 예
8 2 8 10 7 1 9 4 15	1 1 1 3 5 2 5 1

출처: 한국정보올림피아드(2012 전국본선 고등부)

풀이

이 문제를 해결하기 위한 아이디어는 일단 자기보다 앞에서 달리고 있는 선수들 중 자신보다 실력이 높은 사람들의 수를 알아내는 것이다. 자신의 최종 가능한 순위는 현재 자신보다 앞서 달리고 있는 선수들 중 자신보다 실력이 좋은 사람의 수 + 1이 자신이 낼 수 있는 가장 좋은 성적이 될 수 있기 때문이다.

일단 이 문제를 해결하기 위해서 n^2의 계산량으로 쉽게 계산할 수 있다. 자신보다 먼저 입력받은 모든 수들에 대해서 현재 수보다 큰 수들의 개수를 카운팅하면 문제를 해결할 수 있다. 하지만 입력크기 n이 500,000이기 때문에 이 방법으로는 해결할 수 없다.

따라서 IDT 등의 방법으로 해결할 수 있다. 모든 선수들의 실력이 서로 다르다고 했으므로 처음부터 한 명씩 선수를 입력받으면서 해당 선수의 실력에 대한 값에 체크한 후 체크된 값보다 큰 값들의 합을 구하면 앞서 달리던 선수들 중 더 실력이 좋은 선수의 수가 된다.

다음 그림을 통해서 이해해보자. 이 그림은 간단히 배열을 통해서 구현하는 방법에 대해서 안내한다. 먼저 입력값은 2, 8, 10, 7, 1, 9, 4, 15이므로 다음과 같이 채워진다.

	1													
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, 8, 10, 7, 1, 9, 4, 15

처음에 2가 입력되고 2에 체크함.

	1													
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

다음으로 3부터 마지막까지의 합을 구한다. 이 값은 지금까지 입력된 값, 즉 방금 입력된 값보다 먼저 달리고 있던 선수들 중 더 실력이 좋은 선수들의 수가 된다. 따라서 실력이 2인 선수의 최종 순위는 0+1이므로 1위가 된다.

	1						1							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, ~~8~~, 10, 7, 1, 9, 4, 15

다음으로 8이 입력되고 8에 체크함.

	1						1							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

8보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 없으므로 8의 순위도 1위

	1						1		1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, ~~8~~, ~~10~~, 7, 1, 9, 4, 15

다음으로 10이 입력되고 10에 체크함.

	1						1		1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

10보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 없으므로 10의 순위도 1위

	1					1	1		1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, ~~8~~, ~~10~~, ~~7~~, 1, 9, 4, 15

다음으로 7이 입력되고 7에 체크함.

	1					1	1		1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

7보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 8, 9로 2명이 있다.

따라서 7의 순위는 3위가 된다.

1	1					1	1		1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, ~~8~~, ~~10~~, ~~7~~, ~~1~~, 9, 4, 15

다음으로 1이 입력되고 1에 체크함.

1	1					1	1		1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

1보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 모두 4명이 있다.

따라서 1의 순위는 5위가 된다.

1	1					1	1	1	1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, ~~8~~, ~~10~~, ~~7~~, ~~1~~, ~~9~~, 4, 15

다음으로 9가 입력되고 9에 체크함.

1	1					1	1	1	1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

9보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 모두 1명이 있다.

따라서 9의 순위는 2위가 된다.

1	1		1			1	1	1	1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, 8, ~~10~~, ~~7~~, ~~1~~, ~~9~~, ~~4~~, 15

다음으로 4가 입력되고 4에 체크함.

1	1		1			1	1	1	1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

4보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 모두 4명이 있다.

따라서 4의 순위는 5위가 된다.

1	1		1			1	1	1	1					1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

~~2~~, ~~8~~, ~~10~~, ~~7~~, ~~1~~, ~~9~~, ~~4~~, ~~15~~

다음으로 4가 입력되고 4에 체크함.

1	1		1			1	1	1	1					1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

15보다 먼저 달리던 선수들 중 더 실력이 좋은 선수는 모두 0명이 있다.

따라서 15의 순위는 1위가 된다.

이 방법을 이용하면 해를 구할 수 있지만 회색 부분의 합을 구하는데 필요한 계산량은 n이다. 따라서 전체적으로 n^2의 계산량이므로 제대로 해결할 수 없다.

하지만 위 배열을 IDT를 이용하면 계산의 효율을 올릴 수 있다. 즉, 회색 구간의 합을 구하는데 $\lg n$의 시간이 걸리므로 전체적으로 $n \lg n$으로 처리할 수 있다. 그러나 여기에는 또 하나의 문제점이 있다.

위와 같은 방법으로 처리할 경우 선수들의 실력을 나타내는 정수가 최대 10억까지 가능하므로 IDT로 처리하기에는 메모리가 부족하다. 하지만 선수의 수는 50만이므로 적절히 압축하여 실력을 재부여하면 해결할 수 있다. 이를 구현한 소스코드는 다음과 같다.

코드	참고

```
#include <cstdio>
#include <algorithm>

int IDT[1<<21], n, base;
struct P{ int p, s; } S[500001];

bool cmp(P a, P b){ return a.s<b.s;}
```

```
bool rcmp(P a, P b){ return a.p<b.p;}

void update(int v)
{
  while(v)
  {
    IDT[v]=IDT[2*v]+IDT[2*v+1];
    v>>=1;
  }
}

int lg_sum(int a, int b)
{
  int sum=0;
  while(a<b)
  {
    if(a%2==1) sum+=IDT[a], a++;
    if(b%2==0) sum+=IDT[b], b--;
    a>>=1, b>>=1;
  }
  if(a==b) sum+=IDT[a];
  return sum;
}

int main()
{
  scanf("%d", &n);
  for(int i=0; i<n; i++)
    scanf("%d", &S[i].s), S[i].p=i;
  std::sort(S, S+n, cmp);
  for(int i=0; i<n; i++) S[i].s=i;
  std::sort(S, S+n, rcmp);
  for(base=1; base<n; base*=2);
  for(int i=0; i<n; i++)
  {
    IDT[base+S[i].s]=1;
    update((base+S[i].s)>>1);
    printf("%d\n", lg_sum(base+S[i].s+1, base+n)+1 );
  }
  return 0;
}
```

이 소스코드에서 37~39행은 선수들의 능력치를 압축하는 과정이다. 결국 이 값의 순서만 유지하면 구하고자 하는 값과는 관계가 없으므로 이와 같이 처리할 수 있다.

나머지 과정은 IDT를 이용하는 것과 동일하므로 소스코드를 분석해보기 바란다. 다만 미리 입력받고 인덱스를 구성하는 것이 아니라, 값을 입력받으면서 업데이트하며 입력하는 것이 앞에서 다루던 것과는 차이가 있으므로 참고하기 바란다.

Part

III

전 세계적 온라인 대회 참가하기

6. USACO Online Competition
7. Codeforces

6 USACO Online Competition

USACO는 미국에서 운영하고 있는 국제정보올림피아드 트레이닝 사이트로 전 세계적으로 오픈되어 누구나 무료로 이용할 수 있는 사이트로 http://usaco.org로 접속할 수 있다.

이 사이트는 1년에 6회의 정기 대회를 열고, 이 대회를 통하여 미국의 국가대표를 선발한다. 이 대회는 국적에 관계없이 누구나 참여할 수 있다. 메인 페이지 오른쪽에 대회 스케줄을 확인할 수 있다.

USA Computing Olympiad



OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

OUR MISSION

The USACO supports computing education in the USA and worldwide by identifying, motivating, and training high-school computing students at all levels. We provide:

- Hundreds of hours of free on-line training resources that students can use to improve their programming and computational problem-solving skills.
- On-line programming contests (roughly six per year) for students at all levels.
- An intensive summer training camp, to which the top students in the USA are invited to further improve their skills and learn advanced material.
- The opportunity for the top four students in the USA to represent their country at the International Olympiad in Informatics (IOI), the most prestigious international algorithmic programming competition at the high-school level.

TEAM USA WINS SOLID GOLD AT IOI 2014!



Congratulations to team representing the USA at the 2014 International Olympiad in

YOUR ACCOUNT

Not currently logged in.

Username:

Password:

Forgot password?

Login | Register for New Account

2013-2014 SCHEDULE

Nov 15-18: November Contest
Dec 13-16: December Contest
Jan 10-13: January Contest
Feb 7-10: February Contest
Mar 7-10: March Contest
April 4-7: US Open
May 22-31: Training Camp
July 13-20: IOI'14 in Taipei, Taiwan

2013~2014시즌의 대회 스케줄

〈usaco.org 메인 화면〉

가. 가입 방법

홈페이지 우측의 [Register for new Account] 메뉴를 클릭한다.

USA Computing Olympiad

OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

OUR MISSION

The USACO supports computing education in the USA and worldwide by identifying, motivating, and training high-school computing students at all levels. We provide:

- Hundreds of hours of free on-line training resources that students can use to improve their programming and computational problem-solving skills.
- On-line programming contests (roughly six per year) for students at all

YOUR ACCOUNT

Not currently logged in.

Username:

Password:

Forgot password?

Login | Register for New Account

다음으로 각 항목에 값을 채워 넣고 [Submit]버튼을 누른다.

USA Computing Olympiad

OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

CREATE NEW ACCOUNT

Register here for a personalized username for the USACO contest system. Your password will be sent to you immediately via email (check your spam folder if you do not see it right away). You can change your password later by editing your account details. All are welcome to participate in USACO contests and training, including students and non-students, USA and non-USA residents.

Note: Please enter your real name, since this is how you will be identified in our contest results. Fake or unacceptable personal names will not receive contest results and may be deleted at any time without warning!

Username:

Email Address:

First / Given Name: (like John or Jane)

Last / Family Name: (like Smith)

School:

Graduation Year: (set to 9999 if you are past high school / secondary school)

Country: (like USA or CHN; a list is available here)

Submit

Username : 사용하고자 하는 id를 입력한다.

Email Address : 사용할 메일 주소를 입력한다(메일 주소로 비밀번호가 전송되므로 반드시 사용하고 있는 메일 주소를 입력해야 한다.).

First/Given Name : 자신의 이름을 입력한다.

Last/Family Name : 자신의 성을 입력한다.

School : 현재 소속중인 학교를 입력한다.

Graduation Year : 졸업년도를 입력한다(학생일 경우 졸업예정 연도를 입력한다.). 이 값에 따라서 대회 참가 자격이 학생/옵저버로 구분된다.

Country : 자신의 소속국가를 입력한다. 한국의 경우 “KOR”이라고 입력하면 된다.

USACO는 특이하게도 비밀번호를 정하지 않고, 메일을 통하여 받게 된다. 처음 받는 비밀번호가 무작위 문자들로 구성되어 외우기 쉽지 않으므로 처음 로그인하여 비밀번호를 재설정하는 것이 좋다.

나. 비밀번호 재설정하기

USA Computing Olympiad

Overview | Training | Contests | History | Staff | Resources

Our Mission

The USACO supports computing education in the USA and worldwide by identifying, motivating, and training high-school computing students at all levels. We provide:

- Hundreds of hours of free on-line training resources that students can use to improve their programming and computational problem-solving skills.
- On-line programming contests (roughly six per year) for students at all levels.
- An intensive summer training camp, to which the top students in the USA are invited to further improve their skills and learn advanced material.
- The opportunity for the top four students in the USA to represent their country at the International Olympiad in Informatics (IOI), the most prestigious international algorithmic programming competition at the high-school level.

Your Account

Welcome, **Jeong JongKwang**

Edit Account Settings | Logout

2013-2014 Schedule

Nov 15-18: November Contest
Dec 13-16: December Contest
Jan 10-13: January Contest
Feb 7-10: February Contest
Mar 7-10: March Contest
April 4-7: US Open
May 22-31: Training Camp
July 13-20: IOI'14 in Taipei, Taiwan

위의 화면에서 “Edit Account Settiong”를 클릭한다.

USA Computing Olympiad

Overview | Training | Contests | History | Staff | Resources

Edit Account Details

Username: [redacted]
Division:
Email Address:
First / Given Name: (like John or Jane)
Last / Family Name: (like Smith)
School:
Graduation Year: (set to 9999 if you are past high school / secondary school)
Country: (like USA or CHN; a list is available here)

Password:
New Password: (Leave blank if you don't want to change your password)
New Password (Again):

Submit

다음과 같은 설정화면에서 Password 부분에 현재 비밀번호를 입력하고 New Password에 새로 설정할 비밀번호를 2회 입력한 후 [Submit]버튼을 누르면 비밀번호를 재설정할 수 있다.

다. USACO Competiton

USACO는 매년 11월부터 다음해 3월까지 5회의 인터넷 대회를 실시하여 4월에는 오프라인으로 감독관 입회로 US OPEN 대회를 갖는다. 물론 이 대회도 전 세계적으로 공개되는 대회이다.

이들 대회 결과를 바탕으로 15명의 학생을 선발하여 일주일간 UBA Invitational Computing Olympiad 대회를 개최한다. 이때는 미국 국적을 가진 학생들만 초청받아서 진행된다. 왜냐하면 미국의 국가대표는 미국 국적을 가진 학생만 가능하기 때문이다.

일주일간의 치열한 경쟁을 뚫고 선발된 최종 4명의 학생들은 미국 대표팀으로 세계 대회에 출전하게 된다.

US OPEN을 비롯한 6번의 대회는 우리나라 학생들도 참가할 수 있기 때문에 한국정보올림피아드를 준비하는 학생들은 좋은 대회 경험을 가질 수 있다. 대회는 3개의 Division으로 구성된다.

USACO에 처음 가입하면 자동으로 Bronze Division에 속하게 되며, 각 대회가 개최되면 자동으로 Bronze대회에 참가할 수 있다.

Bronze대회에서 일정 기준 이상의 성적을 받게 되면 Silver Division으로 등급이 향상된다. 일반적으로 기준 점수는 1,000점 만점에 800에서 1,000점 사이의 점수를 받아야 된다.

마찬가지로 Silver Division에서 일정 성적 이상을 받게 되면 Gold Division으로 승급하게 된다. Gold Division으로 승급되면 국가대표가 될 수 있는 자격을 가지고 대회에 임할 수 있다고 볼 수 있다.

각 Division은 다음과 같은 특성을 가진다.

- Gold Division

고난이도의 문제를 풀어나갈 수 있는 소수정예 그룹으로 이 그룹의 최하 수준에 있는 학생들도 동적인(dynamic) 알고리즘 프로그래밍을 할 수 있어야 한다.

- Silver Division

Gold Division 바로 아래 수준으로 이 그룹에 속한 학생들은 flood fill 알고리즘과 같은 기본적인 컴퓨터 알고리즘에 익숙해야 한다.

- Bronze Division

이 그룹에 속한 학생들은 복잡하고 이상적인 알고리즘을 요하지 않는 ad hoc 같은 기본적인 프로그래밍을 할 수 있어야 한다.

대회는 3~5시간에 3~4 문제를 풀며 대회 점수가 계속 저조하게 나올 경우 현재보다 아래 단계로 하향 조정도 가능하다. USACO는 코치와 보조 코치들이 있어 90개국의 회원들에게 필요한 정보와 도움을 제공해 주고 있다.

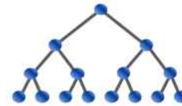
라. USACO Competiton 참가하기

대회 시즌이 되면 대회에 참가할 수 있는 일주일 정도 기간 동안 원하는 시간에 참가할 수 있다. 참가하게 되면 문제의 난이도에 따라 달라지지만 일반적으로 4~5시간의 문제 풀이 시간이 주어진다.

문제 수는 일반적으로 3~4문제가 출제된다. 모든 문제는 Online Judge에 의해 채점되며, 대회가 종료되고 약 일주일 후에 결과와 순위가 공개된다. 한국정보올림피아드에서는 대회장에서 바로 점수를 확인할 수 있으나, USACO에서는 바로 점수를 확인할 수 없기 때문에 정확한 코딩 및 디버깅 능력이 요구된다.

USACO의 결과화면은 다음과 같다.

USA Computing Olympiad



OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

USACO MARCH 2014 CONTEST -- FINAL RESULTS

The USACO March 2014 contest featured algorithmic programming problems covering a wide range of techniques and levels of difficulty.

Click here to see the contest problems and official solutions, or to practice re-submitting solutions.

A total of 1538 participants submitted at least one solution, hailing from 70 different countries:

671 USA	140 CHN	68 IRN	44 VNM	42 BLR	34 GEO	31 CAN
28 TUR	27 IDN	24 MEX	24 DEU	21 KAZ	20 RUS	20 BRA
19 ROU	19 KOR	18 ARM	17 BGR	16 IND	13 SYR	12 BGD
11 TKM	10 VEN	10 FRA	10 BEL	9 THA	9 POL	9 MYS
9 JPN	8 TUN	8 IRL	8 FIN	7 ZAF	7 UKR	7 MNG
7 LTU	7 CUB	6 GRC	6 GBR	5 SGP	5 ITA	5 HRV
5 EST	5 EGY	5 DOM	5 AZE	5 AUS	4 SRB	4 MKD
3 PRT	3 NLD	3 CZE	3 ARG	2 SVN	2 NZL	2 KGZ
2 BIH	2 AUT	1 TWN	1 TJK	1 SVK	1 PER	1 MNE
1 MDA	1 LUX	1 HKG	1 ESP	1 DNK	1 CYP	1 BOL

각 국가별 참가자 인원 등의 간단한 정보가 표시되는 화면이다.

Gold Division Results

[Note: the problem 'fcount' has been re-graded after fixing a minor issue with the test data]

Full Gold Results

The Gold division had 350 total participants, of whom 258 were pre-college students.

Top pre-college participants:

Country	Grad	Name	Score
KOR	2015	Seokhwan Choi	1000
VNM	2015	Van Hanh Pham	1000
VNM	2015	Khanh Do Ngoc	1000
USA	2014	Steven Hao	1000
USA	2015	Scott Wu	1000
CHN	2014	Mingda Qiao	1000
USA	2015	Andrew He	1000
USA	2016	Demi Guo	1000
USA	2014	Jerry Ma	1000
USA	2014	Qingqi Zeng	1000
POL	2014	Michał Glapa	1000
KOR	2016	Seunghyun Joe	1000
USA	2014	Joshua Brakensiek	1000
IRN	2015	Keivan Mohtashami	1000

각 Division별로 우승자의 국가 및 이름이 화면에 표시된다. Gold Division에서 1,000점 만점을 획득한 사람이 14명이 있고 그 중 2명이 한국의 학생들이다.

Full Gold Results를 클릭하면 전체 상세 결과를 볼 수 있다.

Final Results: USACO 2014 March Contest, Gold

Key:
* = Correct
x = Wrong Answer
t = Timeout
c = Didn't Compile
s = Signal (crashed, exceeded memory limits, invalid syscall)

Pre-College Participants (Full List):

Country	Year	Name	Score	lazy	sabotage	fcount
KOR	2015	Seokhwan Choi	1000	***********	**************	***********
VNM	2015	Van Hanh Pham	1000	***********	**************	***********
VNM	2015	Khanh Do Ngoc	1000	***********	**************	***********
USA	2014	Steven Hao	1000	***********	**************	***********
USA	2015	Scott Wu	1000	***********	**************	***********
CHN	2014	Mingda Qiao	1000	***********	**************	***********
USA	2015	Andrew He	1000	***********	**************	***********
USA	2016	Demi Guo	1000	***********	**************	***********
USA	2016	Tony Wang	455	*ssssssssss	**************	*xxxxxxx*x*
USA	2014	Brian Shimanuki	455	*tttttttttt	**************	*xxx*ttttt*
USA	2015	Brice Huang	429		*x*xxxxxxxx*x*	***********
BGD	2014	Bristy Sikder	424	***********		*xxxxxxx*x*
USA	2015	Khanh Chau Bui	411	*tttttttttt	*xxxxx*xxtxttt	***********
ARM	2014	Sparik Hayrapetyan	405	*tttttttttt	*x*xxxxx*xxxxx	***x*******
MEX	2014	Daniel Talamas	405	***********	**xxxx*xxxxxxx	
USA	2015	Alexander Dai	403		**xx*x****xx*x	****tttt***
USA	2016	Yang Yan	387	*xxxxxxxxxx	*xxxxxxxxttttt	***********

각 참가자들의 전체 성적을 상세히 볼 수 있는 페이지이다. 위의 예에서 표시된 학생에 대해서 살펴보면 왼쪽 칸으로부터, 참가국가, 졸업년도, 참가자 이름, 획득 점수, 각 3문제에 대한 결과를 의미한다.

각 문제에 대한 결과는 다음과 같은 의미를 지닌다.

표시문자	의미
*	맞았음(해당 테스트 케이스에 대해서 정확한 답을 제한시간 이내에 출력했음을 의미한다.)
x	틀렸음(해당 테스트 케이스에 대해서 제한 시간 이내에 답을 출력했으나, 원래 요구하는 답과 틀린 답을 출력한 경우)
s	signal out(입력값을 받아서 프로그램이 실행되다 알 수 없는 이유로 프로그램이 종료된 경우, 일반적으로 런타임 에러일 경우가 많으며 대부분 배열의 잘못된 사용, 0으로 나누기 등일 확률이 높다.)
t	시간초과(해당 테스트 케이스에 대해서 주어진 제한시간 이내에 알고리즘이 종료되지 못한 경우, 일반적으로 알고리즘의 효율이 너무 좋지 않았을 경우가 많다.)

마. USACO Competiton 기출문제 풀어보기



먼저 USACO페이지에서 [Contest] 메뉴를 선택한다.



다음으로 원하는 이전 대회 중 하나를 클릭한다.



다음으로 [Click here]를 클릭하면 문제를 풀어볼 수 있는 페이지로 이동한다.

USA Computing Olympiad

OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

USACO 2013 DECEMBER CONTEST, GOLD

1 Vacation Planning (gold)
View problem | Test data | Solution

2 Optimal Milking
View problem | Test data | Solution

3 The Bessie Shuffle (gold)
View problem | Test data | Solution

USACO 2013 DECEMBER CONTEST, SILVER

1 Milk Scheduling
View problem | Test data | Solution

2 Vacation Planning
View problem | Test data | Solution

3 The Bessie Shuffle
View problem | Test data | Solution

USACO 2013 DECEMBER CONTEST, BRONZE

1 Record Keeping
View problem | Test data | Solution

YOUR ACCOUNT

Welcome, **Jeong JongKwang**

Edit Account Settings | Logout

원하는 문제를 클릭한다. 일반적으로 처음에는 Bronze를 클릭하여 연습하는 것이 좋다.

USA Computing Olympiad

OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

USACO 2013 DECEMBER CONTEST, BRONZE
PROBLEM 1. RECORD KEEPING

Return to Problem List
Contest has ended.

Analysis mode

Problem 1: Record Keeping [Brian Dean, 2013]

Lang: en

Farmer John has been keeping detailed records of his cows as they enter the
barn for milking. Each hour, a group of 3 cows enters the barn, and Farmer
John writes down their names. For example over a 5-hour period, he might
write down the following list, where each row corresponds to a group
entering the barn:

BESSIE ELSIE MATILDA
FRAN BESSIE INGRID
BESSIE ELSIE MATILDA
MATILDA INGRID FRAN
ELSIE BESSIE MATILDA

Farmer John notes that the same group of cows may appear several times on
his list; in the example above, the group of BESSIE, ELSIE, and MATILDA

다음과 같이 문제를 볼 수 있다. 화면 상단에 Analysis mode라고 표시되며, 이 경우 답안을 제출하여 결과를 확인할 수 있다.

OUTPUT DETAILS:

The group {BESSIE, ELSIE, MATILDA} enters the barn on three separate occasions.

Language: C++ 1

Source File: 파일 선택 선택된 파일 없음 2

Submit Solution 3

Note: Many issues (e.g., uninitialized variables, out-of-bounds memory access) can cause a program to product different output when run multiple times; if your program behaves in a manner inconsistent with the official contest results, you should probably look for one of these issues. Timing can also differ slightly from run to run, so it is possible for a program timing out in the official results to occasionally run just under the time limit in analysis mode, and vice versa.

문제를 해결한 후에는 C/C++언어로 코딩을 하고 문제 하단부에 있는 그림과 같은 부분의 1, 2, 3의 순서대로 파일을 선택하고 제출할 수 있다. 제출한 후에는 그 결과를 다음과 같이 확인할 수 있다.

USA Computing Olympiad

Overview | Training | Contests | History | Staff | Resources

USACO 2013 December Contest, Bronze
Problem 1. Record Keeping

Return to Problem List
Contest has ended.

Submitted

1	2	3	4	5	6	7	8	9	10
* 11.3mb 2ms	* 11.3mb < 1ms	* 11.3mb 1ms	* 11.3mb 1ms	* 11.3mb 2ms	* 11.3mb 9ms	* 11.3mb 20ms	* 11.3mb 21ms	* 11.3mb 25ms	* 11.3mb 47ms

Problem 1: Record Keeping [Brian Dean, 2013]

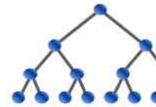
Lang: en

Farmer John has been keeping detailed records of his cows as they enter the barn for milking. Each hour, a group of 3 cows enters the barn, and Farmer John writes down their names. For example over a 5-hour period, he might write down the following list, where each row corresponds to a group entering the barn:

앞에서 analysis mode라고 표시되었던 부분에 채점 결과를 볼 수 있다. 이와 같이 문제들에 대해서 연습할 수 있다.

그 이외에 테스트 데이터와 풀이 등의 자료도 공개하고 있으므로 공부하는데 많은 도움이 된다. 이 문제에 대한 풀이는 다음과 같이 확인할 수 있다.

USA Computing Olympiad



OVERVIEW | TRAINING | CONTESTS | HISTORY | STAFF | RESOURCES

USACO 2013 DECEMBER CONTEST, GOLD

1 Vacation Planning (gold)
View problem | Test data | Solution

2 Optimal Milking
View problem | Test data | Solution

3 The Bessie Shuffle (gold)
View problem | Test data | Solution

USACO 2013 DECEMBER CONTEST, SILVER

1 Milk Scheduling
View problem | Test data | Solution

2 Vacation Planning
View problem | Test data | Solution

3 The Bessie Shuffle
View problem | Test data | Solution

USACO 2013 DECEMBER CONTEST, BRONZE

1 Record Keeping
View problem | Test data | Solution

YOUR ACCOUNT

Welcome, **Jeong JongKwang**

Edit Account Settings | Logout

[Solution]을 클릭한다. 만약 [Test data]를 클릭하면 채점용 데이터를 다운 받을 수 있다.

Analysis: Record Keeping by Fatih Gelgi

The trivial idea is to try all possibilities: check how many times each group appears in the list. We know that the order of cows can be different in same groups. Trying all cow permutations in a group complicates coding. Instead, a better idea is to sort the cows alphabetically in a group and store them in a string. Then we can search each string in the list easily. For instance, the sample will input will be as follows after sorting each group:

```
BESSIE ELSIE MATILDA
BESSIE FRAN INGRID
BESSIE ELSIE MATILDA
FRAN INGRID MATILDA
BESSIE ELSIE MATILDA
```

Now, we can clearly see the string "BESSIE ELSIE MATILDA" appears in the list three times. Note that we put space between cows instead of having "BESSIEELSIEMATILDA". Otherwise, the solution will fail in the following input - "BESSI EELSIE MATILDA" is not same as "BESSIE ELSIE MATILDA":

```
BESSI EELSIE MATILDA
BESSIE FRAN INGRID
BESSIE ELSIE MATILDA
FRAN INGRID MATILDA
BESSIE ELSIE MATILDA
```

The running time will be O(N^2) since we search each string in the entire list. N is small hence the solution is fast enough for the problem. Sample code is as follows:

```
// O(N^2)
#include <fstream>
#include <algorithm>
```

위 화면은 풀이의 일부이다. 간단한 알고리즘 소개와 해법 소스코드를 공개한다. Bronze Division의 경우 일부 문제는 Youtube 풀이 동영상을 함께 공개하는 경우도 있으므로 학습하는데 많은 도움이 된다.

7 Codeforces

코드포스는 러시아에서 운영하고 있는 프로그래밍 콘테스트 사이트로 러시아어와 영어만을 지원한다. 평균적으로 일주일에 1번의 대회가 열리는 사이트로 한 번 대회가 열릴 때 3,000명 이상의 참가자가 모이는 매우 큰 규모의 대회이다.

코드포스는 http://codeforces.com으로 접속할 수 있으며 메인화면은 다음과 같다.

가. Codeforces에 가입하기

다음 그림과 같이 메인 화면 오른쪽 상단의 [Register]을 눌러 회원 가입할 수 있다.

가입 시에는 다음 그림과 같이 3가지 정보만 있으면 가입할 수 있다. 그리고 Gmail ID가 있으면 바로 이용할 수 있으므로 편리하다.



Enter | Register

HOME CONTESTS GYM PROBLEMSET GROUPS RATING API HELP BAYAN 2015 RCC

Fill in the form to register in Codeforces.
You can skip this step and login with your OpenID or Gmail account.

Register in Codeforces

Handle: KOI
Choose your username (nickname) on Codeforces. Be careful, you will not be able to change it later.

Email

Password
Password should contain at least five characters

Confirm Password

Register

Re-send Confirmation Email | Use OpenID | Use Gmail

Handle : 일반적으로 ID에 해당하며, 특별한 이벤트가 있으면 이 handle을 바꿀 수 있는 경우도 있다.

Email : 자신의 메일 주소를 쓴다. 이 메일 주소로 대회 공지 등의 안내가 있으므로 실제 사용가능한 주소를 쓸 수 있도록 한다.

Password : 비밀번호이다. 2번 반복하여 쓴다.

가입한 후 로그인 하면 프로필 화면으로 이동할 수 있다. 프로필 화면에서는 해당 handle을 가진 참가자의 대회 참가 이력을 비롯하여 각종 정보를 제공해준다.

Grandmaster
gs12117
Seokhwan Choi, Seongnam, Korea, Republic of
From Gyeonggi Science High School

Contest rating: 2386
(max. grandmaster, 2386)

Contribution: +7

sukh1222@naver.com

Last visit: 19 hours ago

Registered: 23 months ago

Blog entries (0), comments

Send message to gs12117



현재 gs12117인 handle을 가진 사람의 프로필을 볼 수 있다. 가장 먼저 나오는 Grandmaster라는 것은 해당 handle의 칭호를 나타낸다. 칭호는 다음과 같은 단계로 구성된다.

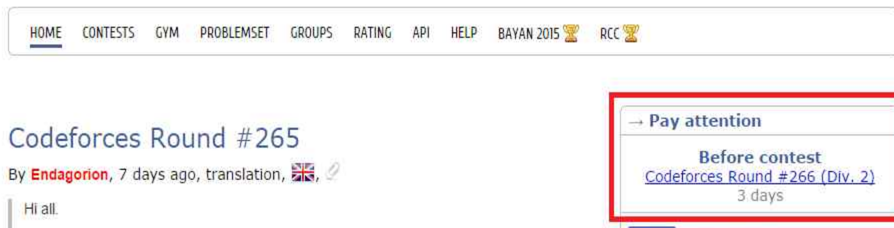
Division	칭호	Rating
Div 1	International Grandmaster	2600~
	Grandmaster	2200~2599
	International master	2050~2199
	master	1900~2049
	Candidate master	1700~1899
Div 2	Expert	1500~1699
	Specialist	1350~1499
	Pupil	1200~1349
	Newbie	~1199

위 표와 같이 codeforces는 대회의 성적에 따라서 자신의 역량을 나타내는 Rating이라는 점수로 칭호를 부여한다. 처음 참가할 때는 0점이지만 실제로 1500점을 기준으로 점수가 산정된다.

그리고 Rating에 따라서 Div1과 Div2로 나누어 대회를 따로 치르며, 대회의 성적에 따라 Rating이 바뀌게 된다.

나. 대회에 참가하기

Codeforces에 처음 참가하면 Div2에 참가할 수 있다. 참가 신청은 대회 24시간 전에 할 수 있으며, 대회시작 5분 전까지 참가신청이 가능하다.



앞의 그림과 같이 오른쪽 상단에 현재 준비 중인 대회가 표시된다. 대회를 클릭하고 참가신청을 하면 된다. 단, 대회시작 5분 전까지만 신청할 수 있으므로 유의해야 한다.

CODEFORCES β
Sponsored by Telegram

sutekine | Logout

HOME CONTESTS GYM PROBLEMSET GROUPS RATING API HELP BAYAN 2015 RCC

Codeforces Round #266 (Div. 2)

Name	Start	Duration		
Codeforces Round #266 (Div. 2)	Sep/12/2014 19:30	02:00	Before start 3 days	Before registration 3 days

* To view the complete list, click the link.

Like 81 people like this. Tweet 1

Before the end of the registration
3 days

Before the contest
3 days

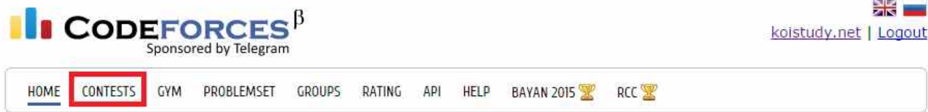
위 화면에서 참가 신청을 하면 참가할 수 있다.

Codeforces에서는 이미 종료된 정규대회에 대한 가상대회를 지원한다. 가상대회란 실제 대회를 저장해두었다가 가상참가자들을 실제 대회에 추가하여 참가할 수 있는 기회를 제공하는 것이다.

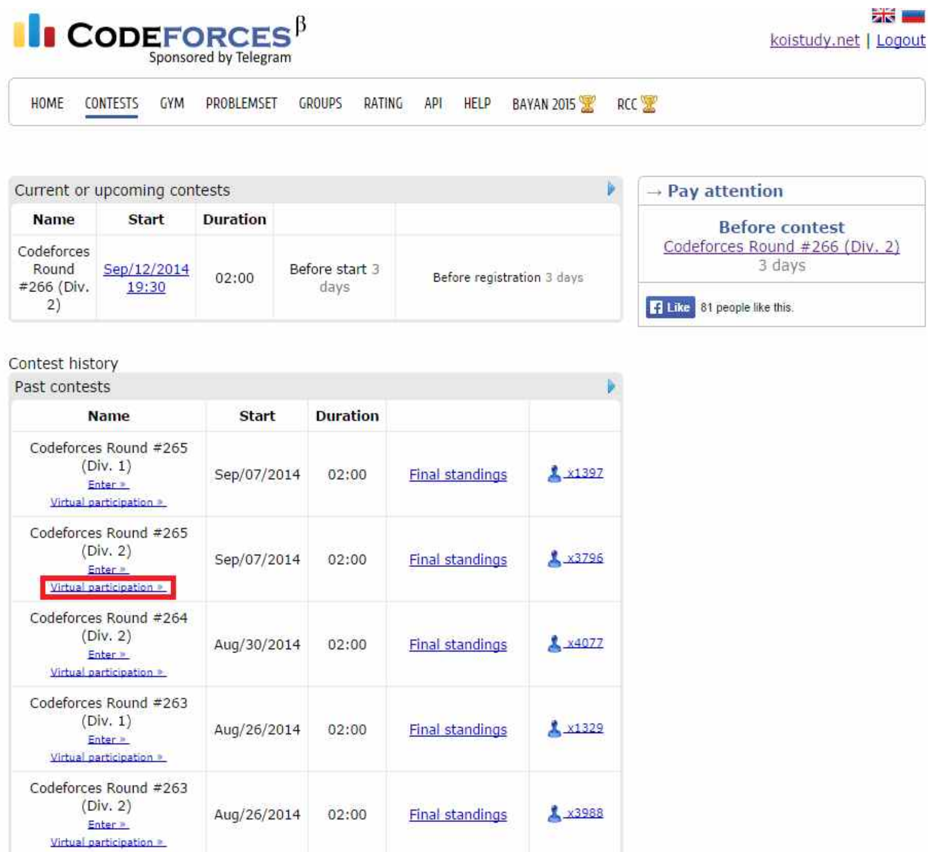
가상참가를 하면 실제 대회와 같이 진행되며 실제 대회의 순위표에 가상참가자를 추가하여 함께 표시되기 때문에 실제 대회의 느낌을 받을 수 있다. 다만 진행 중에 점수 부여방식은 Codeforces방식이 아니라 ACM ICPC방식으로 바뀌기 때문에 긴장감은 다소 떨어진다.

거의 새벽에 열리는 본 대회에 참가가 어렵다면 가상대회에 참가하는 것도 좋은 경험이 될 것이다. 가상대회에 참가하는 것으로는 Rating의 변화는 없다.

가상대회에 참가하는 방법은 다음과 같다.



먼저 메인 메뉴에서 [Contests]를 선택한다.



다음으로 원하는 대회의 Virtual participation(가상참가)을 선택한다.

Registration for virtual participation

Codeforces Round #265 (Div. 2)

Notice: virtual participation

Terms of agreement:

Virtual contest is a way to take part in past contest, as close as possible to participation on time. It is supported only ACM-ICPC mode for virtual contests.

If you've seen these problems, a virtual contest is not for you - solve these problems in the archive.

If you just want to solve some problem from a contest, a virtual contest is not for you - solve this problem in the archive.

Never use someone else's code, read the tutorials or communicate this other person during a virtual contest.

Take part: as individual participant / as a team member

Virtual start time: Sep/09/2014 10:35 — 1

Use datetime format like 03/07/2010 11:00PM

[Register for virtual participation] — 2

가상참가에서 원하는 시간을 선택하고 [Register for virtual participation]을 선택하면 원하는 시간에 대회가 시작된다.

Codeforces에서는 5분마다 가상대회를 열 수 있도록 지원할 수 있으며, 주변의 친구들과 함께 가상대회에 참가하면 좋은 경험을 할 수 있다.



koistudy.net | Logout

HOME CONTESTS GYM PROBLEMSET GROUPS RATING API HELP BAYAN 2015 RCC — Search by tag

PROBLEMS SUBMIT CODE MY SUBMISSIONS STATUS STANDINGS CUSTOM INVOCATION

Problems

#	Name		
A	inc ARG	standard input/output 1 s, 256 MB	 x2
B	Inbox (100500)	standard input/output 1 s, 256 MB	
C	No to Palindromes!	standard input/output 1 s, 256 MB	
D	Restore Cube	standard input/output 1 s, 256 MB	
E	Substitutes in Number	standard input/output 1 s, 256 MB	

Complete problemset

Codeforces Round #265 (Div. 2)
Contest is running
01:59:09
Virtual Participation

다음은 대회 진행화면이다. 대회는 5개의 문제가 준비되어 있으며, 각 문제를 클릭하여 읽은 후 소스코드를 제출하면 서버에서 자동 채점하여 그 결과가 순위에 반영되는 방법이다. 직접 한 번 경험해 보는 것이 가장 좋은 방법이므로 기회가 되면 참가해보기 바란다.

다. 대회에서 점수 산정 방법

Codeforces의 실제 대회에서 점수는 다음과 같은 방법으로 산정된다.

문항 번호	초기 점수	time penalty(1min)
1	500	2 pts
2	1000	4 pts
3	1500	6 pts
4	2000	8 pts
5	2500	10 pts

위 표는 일반적인 대회에서의 점수 산정 방법이다. 기본적으로 뒤 번호의 문항일수록 점수가 크다. 그리고 시간 감점이라는 개념이 도입되기 때문에 이 대회에서는 똑같이 3문제를 풀더라도 더 빨리 해결한 사람의 점수가 높다.

만약 어떤 참가자가 10분 만에 1번 문제를 풀었다면 480점을 획득하게 된다. 즉 10분 동안 1분에 2점씩 감점되어 480이 된다. 다른 참가자는 1번 문제를 12분 만에 해결했다면 476점이 되기 때문에 대부분의 참가자들이 동점자가 발생하지 않는다.

그리고 어려운 문제일수록 더 많은 점수가 감점되므로 여러 가지 전략을 세워서 대회에 임하는 것이 고득점을 노리는데 유리하다.

다음으로 대회 중 각종 감점 및 가산점에 대해서 소개하고자 한다.

보너스 점수	점수	내용
실패 감점	−50	제출한 답안이 틀렸을 때, 하지만 다시 재출 가능함.
해킹 가점	+100	상대방의 소스코드의 오류를 잡아 잡아내려고 한 시도가 성공했을 경우
해킹 감점	−50	상대방의 소스코드의 오류를 잡아내려고 한 시도가 실패했을 경우

Codeforces에서는 대회 중에 Hack이라고 하는 메뉴가 등장한다. 이 메뉴는 다른 참가자의 소스코드를 분석하여, 오류가 있다는 것을 검증하는 시도이다. 따라서 특정 테스트 케이스를 제작하여 Hack을 시도할 수 있다.

만약 시도한 Hack이 성공했다면, 즉 만든 테스트 케이스에 대해서 상대방의 코드가 오답을 출력한다면 100점을 획득할 수 있다. 만약 실패한다면 50점의 감점을 받게 된다.

Hack은 매우 다양한 변수를 제공하며 대회의 재미를 배가시키는 역할을 한다. 그리고 상대방의 소스코드를 정확하게 분석할 줄 알아야하므로 코딩 실력을 향상시키는 데 큰 도움을 준다.

Hack에 대한 자세한 사항은 Codeforces를 어느 정도 참가하다보면 자연스럽게 익숙해지므로 여기서는 더 이상 설명하지는 않는다.